УДК 519.683.8

ИМПОРТ ВЫЧИСЛИТЕЛЬНОЙ МОДЕЛИ ЯЗЫКА SCHEME В ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ОКРУЖЕНИЕ

© 2008 г. А. В. Столяров

avst@cs.msu.ru

Кафедра Алгоритмических языков

1 Введение

Метод непосредственной интеграции впервые предложен в статье [1] в качестве подхода к интеграции разнородных языковых изобразительных средств в одном проекте. Методу также посвящены работы [2], [3], [4], [6] и [7].

В основе метода непосредственной интеграции лежит моделирование средствами базового языка одновременно вычислительной модели (семантики) и изобразительных конструкций (синтаксиса) альтернативного языка. При этом синтаксис альтернативного языка моделируется близкими по визуальному восприятию арифметическими выражениями, записываемыми в соответствии с правилами языка базового. Это достигается введением объектов, соответствующих понятиям альтернативного языка и переопределением символов инфиксных операций для таких объектов; ясно, что для применения метода необходимо наличие в базовом языке соответствующих возможностей.

Метод практически реализован в библиотеке InteLib, которая предоставляет набор классов языка C++, реализующих S-выражения [8] как гетерогенные структуры данных. Вводимый для этих классов набор инфиксных операций позволяет записывать конструкции, являющиеся с точки зрения компилятора C++ арифметическими выражениями, которые синтаксически близки традиционным для языка Lisp S-выражениям. Второй слой библиотеки позволяет производить *вычисления* этих выражений, достигая, таким образом, их семантической эквивалентности соответствующим выражениям языка Lisp.

2 Проблемы реализации модели языка Scheme

Исходно вычисления конструкций языка Lisp в библиотеке InteLib были реализованы простейшим для понимания способом, а именно, с использованием рекурсивного вызова программы вычисления выражения. Так, например, при вычислении

```
(+ 1 (* 2 x) (* 3 x x))
```

вычислитель сначала вызывался для всего выражения целиком. Поскольку для передачи управления функции «+» необходимо сначала вычислить её арнументы, вычислитель вызывал рекурсивно сам себя для вычисления (поочерёдно) выражений `1`, `(* 2 x)` и `(* 3 x x)`. Будучи вызванным для константы `1`, он немедленно возвращал управление, поскольку действие по вычислению константы является элементарным. Будучи далее вызванным для `(* 2 x)`, вычислитель вновь рекурсивно вызывал сам себя уже для выражений `2` и `x`; в первом случае управление возвращалось немедленно, во втором — анализировался текущий лексический контекст, и возвращалось значение, лексически связанное с `x` (то есть локальное), если таковое присутствовало, в противном случае возвращалось динамическое (глобальное) значение `x`. Получив все значения, необходимые для вызова функции «*», вычислитель вызывал тело функции, предварительно сформировав вектор из её фактических параметров, получал значение, возвращал управление, и т. д.

В первой (пробной) версии библиотеки InteLib, написанной в 1999 году, результат работы вычислителя возвращался как значение функции. Для оптимизации остаточной рекурсии, а также для реализации формы `SETF` механизм возврата значений был усовершенствован уже во

второй версии: в функцию, вычисляющую S-выражение, стал передаваться дополнительный параметр, представляющий собой объект специального класса, который отвечал за переключение лексических контекстов, восстановление значений динамически связываемых переменных и хранение возвращаемого значения, причём возврат результата вычисления стало возможно производить в соответствии с одной из трёх моделей: вернуть обычное значение, вернуть *ссылку* на имеющуюся структуру данных (например, на левую или правую часть существующей точечной пары), либо вернуть выражение, которое в свою очередь подлежит вычислению. Вторая из перечисленных моделей сделала возможной реализацию `SETF`, третья — оптимизацию остаточной рекурсии. Так или иначе, для вычисления аргументов функции, а также для вызова других функций в функциях высоких порядков (таких как `MAPCAR` или `SORT`) вычислитель продолжал делать рекурсивные вызовы самого себя.

Для вычислительной модели языка Scheme такая реализация оказывается неприемлема из-за наличия в этом языке механизма *продолжений* (continuations), реализуемого функцией `CALL-WITH-CURRENT-CONTINUATION` (или просто `CALL/CC`). С теоретической точки зрения *продолжение* представляет собой *последовательность действий, которые необходимо выполнить до завершения вычислений.* Каждое элементарное действие, такое как вычисление константы или применение функции к вычисленным аргументам, рассматривается как *переход* от одного *продолжения* к другому; иногда также говорят, что новое (очередное) продолжение возникает в тот же момент, когда начинается выполнение очередного элементарного вычисления, и *ожидает значения.* Так, например, при вычислении

```
(+ (* a 15) (* b 25))
```

первое *продолжение* имеет вид «вычислить выражение целиком»; от него осуществляется переход к *продолжению* «получить значение, затем вычислить `(* b 25)`, затем к двум последним значениям применить `+`». Можно сказать, что это второе *продолжение* возникает в тот же момент, когда начинается вычисление `(* a 15)`, и ожидает его результата.

Функция `CALL/CC` позволяет «законсервировать» текущее *продолжение*, с тем чтобы позже можно было выполнить те же действия, но начальный результат дать другой.

Если рассматривать описанную выше простую рекурсивную реализацию вычислителя, окажется, что информация, составляющая *продолжение*, рассеяна по локальным переменным многих одновременно активных экземпляров функции-вычислителя, и, более того, часть информации вообще не является явно доступной: действительно, информация о предстоящей последовательности действий оказывается представлена совокупностью адресов возврата, сохранённых в стеке, которые в явном виде программисту недоступны, во всяком случае, если говорить о штатных переносимых средствах. Известна реализация механизма *продолжений* для языка C, основанная на недокументированных особенностях функций `setjmp()` и `longjmp()` (см. [11]), однако гарантировать переносимость такого решения невозможно.

3 Идея применённого решения

Как можно заметить, при работе с *продолжением* речь идёт о двух основных сущностях: о действиях, которые необходимо выполнить, и о сохранённых значениях, которые являются результатами уже выполненных действий.

Легко видеть также, что при выполнении вычислений бóльших уровней вложенности одни действия по мере их выполнения могут заменяться другими либо преобразовываться в готовые значения, причём чем позднее было получено значение, тем раньше оно понадобится при вызове очередной функции; аналогично, чем позднее в список невыполненных действий попала та или иная операция, тем раньше настанет момент, когда её придётся выполнить.

Получается, что *продолжение* можно представить в виде двух стеков: в одном будем сохранять операции, которые необходимо произвести, во втором — значения, которые позднее станут аргументами операций. Для примера рассмотрим процесс вычисления выражения `(+ (* a 15) (* b 25) 7)` при значениях переменных `a` и `b`, соответственно, `10` и `100`. Последовательность шагов (*продолжений*) такого вычисления показана в табл. 1.

При этом мы обозначаем словом `EVAL` операцию, состоящую в вычислении заданного выражения, а словом `CALL` — применение заданной функции к заданному количеству аргументов.

№ шага	Стек значений	Стек операций
1	∅	`EVAL (+ (* a 15) (* b 25) 7)`

2	∅	`EVAL (* a 15)`
		`EVAL (* b 25)`
		`EVAL 7`
		`CALL/3 +`

3	∅	`EVAL a`
		`EVAL 15`
		`CALL/2 *`
		`EVAL (* b 25)`
		`EVAL 7`
		`CALL/3 +`

4	10	`EVAL 15`
		`CALL/2 *`
		`EVAL (* b 25)`
		`EVAL 7`
		`CALL/3 +`

5	15
	10
		`CALL/2 *`
		`EVAL (* b 25)`
		`EVAL 7`
		`CALL/3 +`

6	150	`EVAL (* b 25)`
		`EVAL 7`
		`CALL/3 +`

7	150	`EVAL b`
		`EVAL 25`
		`CALL/2 *`
		`EVAL 7`
		`CALL/3 +`

8	100
	150
		`EVAL 25`
		`CALL/2 *`
		`EVAL 7`
		`CALL/3 +`

9	25
	100
	150
		`CALL/2 *`
		`EVAL 7`
		`CALL/3 +`

10	2500 150	`EVAL 7` `CALL/3 +`
11	7 2500 150	`CALL/3 +`
12	2657	∅

Таблица 1: Пример последовательности *продолжений*

Первая операция выполняется за один шаг для констант и переменных, т.е. операция убирается из стека действий, а в стек результатов добавляется значение. Для вызовов функций операция `EVAL` выполняется иначе: сама она из стека действий убирается, а вместо неё заносится операция `CALL` для соответствующего количества параметров и операции `EVAL` для вычисления каждого аргумента функции; стек результатов при этом не меняется. Наконец, операция `CALL` извлекает из стека результатов соответствующее количество параметров, применяет к ним заданную функцию, а затем результат применения помещает снова в стек результатов.

Шаги, перечисленные в таблице, соответствуют в теоретической терминологии последовательности *продолжений*. Вычисления считаются оконченными, когда стек действий пуст, то есть *продолжение* с пустым стеком действий считается конечным. К этому моменту в сте-

ке результатов должно остаться ровно одно значение, которое и объявляется окончательным результатом вычисления.

Ясно, что в сохранении всех промежуточных *продолжений* нет никакого практического смысла, если только не выполняются функции, «консервирующие» текущее *продолжение*. Таким образом, достаточно иметь *один* экземпляр стеков и изменять их, а при необходимости для целей «консервирования» создавать копии.

При реализации в объектно-ориентированном окружении естественной выглядит инкапсуляция обоих стеков в класс, который и называется `Continuation` (т.е. «продолжение»), причём такой объект должен иметь методы по добавлению новых заданий, по извлечению результатов, а также метод, выполняющий *один шаг вычислений*, или, иначе говоря, *переход к следующему продолжению*.

Объект такого класса может при желании рассматриваться и как своего рода виртуальная машина, исполняющая программы на языке Scheme.[1]

Поскольку объект, представляющий *продолжение*, неизбежно должен существовать и быть доступным во всём коде, имеющем отношение к вычислению S-выражений (исполнению программы на Scheme), на этот объект можно также возложить некоторые вспомогательные функции, такие как управление связыванием локальных переменных (иначе говоря, управление «лексическими контекстами») и поддержку различных моделей возврата значения из функции.

4 Рабочая реализация

4.1 Предварительные замечания

4.1.1 Структура библиотеки InteLib

Библиотека InteLib спроектирована в несколько слоёв, причём нижний слой представляет собой S-выражения, воспринимаемые как гетерогенные структуры данных в отрыве от каких-либо вычислительных моделей [6].

Верхний слой библиотеки содержит подсистемы, относящиеся к вычислительным моделям различных языков. В настоящее время дистрибутив библиотеки содержит модели языков Lisp, Scheme и Refal, причём последняя пока находится в стадии экспериментального кода. Ведутся также работы над моделями языков Planner, Prolog, Datalog и Erlang. Необходимо отметить общее для всех этих языков свойство: все они используют гетерогенные структуры данных, основанные на односвязных списках, т.е. S-выражения, хотя при описании этих языков термин «S-выражение» почему-то обычно не употребляется.

Кроме верхнего и нижнего слоёв, библиотека имеет также промежуточный слой, в котором реализованы некоторые инструментальные средства для работы с S-выражениями, в частности — средства работы с потоками ввода-вывода, синтаксического анализа текстового представления S-выражений и т.п. Также в этот слой вынесены общие для различных вычислительных моделей части, имеющие отношение к *вычислению* S-выражений.

Библиотека ориентирована на работу в «чисто компилируемом» окружении и использование арифметических выражений C++, заменяющих соответствующие конструкции моделируемых языков. Тем не менее, в мультипарадигмальном окружении практически всегда возникает настойчивая потребность в написании некоторых модулей целиком в синтаксисе оригинального (импортируемого) языка. В дистрибутив библиотеки InteLib входят трансляторы для каждого из моделируемых языков программирования, позволяющие на основании текста на таком языке построить модуль на C++, состоящий из заголовочного файла и файла реализации и использующий конструкции библиотеки InteLib. Кроме того, для целей отладки в библиотеку входят и расширяемые интерактивные интерпретаторы языков Lisp и Scheme; при необходимости эти интерпретаторы можно включать и в пользовательские приложения в качестве встроенных.

[1] Естественно, здесь имеется в виду внутреннее представление программ, поскольку задачи лексического и синтаксического анализа кода относятся к совершенно иной проблемной области.

4.1.2 Соглашения об именованиях классов в библиотеке InteLib

Отметим, что в библиотеке InteLib S-выражения различных типов представляются объектами классов, имеющих общего абстрактного предка `SExpression`. Имена классов, представляющих конкретные типы S-выражений и относящихся к нижнему слою библиотеки, начинаются с префикса `SExpression`; такое же соглашение используется для классов промежуточного (инструментального) слоя.

Классы S-выражений, относящихся к конкретным вычислительным моделям (Lisp, Scheme, Refal) имеют в названии префикс, указывающий на соответствующую вычислительную модель (соответственно префиксы `LExpression`, `SchExpression` и `RfExpression`).

Работа с S-выражениями строится через систему «умных указателей» и «ведущих указателей» (*smart pointers* и *master pointers* в терминологии книги [12]). Умные указатели — это объекты, ведущие себя подобно указателям, но выполняющие какие-либо дополнительные функции; в случае библиотеки InteLib наиболее важная из таких функций — поддержка счётчиков ссылок, позволяющая автоматически уничтожать объекты S-выражений, на которые больше никто не ссылается. Кроме того, на умные указатели возлагается ряд функций, специфичных для S-выражений, таких как композиция и декомпозиция списков, конверсия во встроенные типы и т.д. Все умные указатели в библиотеке InteLib наследуются от класса `SReference`, который представляет собой основной механизм доступа к S-выражениям. Классы умных указателей, вводимые для обслуживания специфических типов S-выражений, имеют имена, полученные из названий соответствующих классов S-выражений путём удаления слова `Expression` и добавления суффикса `Ref`: например, хеш-таблицы в InteLib представляются классом `SExpressionHashTable`, а соответствующий им класс умного указателя имеет имя `SHashTableRef`; аналогично, *символ* языка Scheme представляется классом `SchExpressionSymbol`, а соответствующий класс умного указателя называется `SchSymbolRef`.

Ведущие указатели отличаются от умных указателей тем, что сами автоматически создают объект, на который указывают. В библиотеке InteLib классы ведущих указателей всегда наследуются от соответствующих умных указателей и имеют имена, полученные отбрасыванием суффикса `Ref`; так, ведущие указатели для хеш-таблиц называются `SHashTable`, а для символов языка Scheme — `SchSymbol`.

Для обработки исключительных ситуаций используется класс, называемый `IntelibX` и его потомки, названия которых формируются добавлением к префиксу `IntelibX` слов, обозначающих конкретную ошибочную ситуацию, причём эти слова набираются в нижнем регистре через символ подчёркивания; например, арифметические функции при подаче им нечисловых аргументов выбрасывают исключение `IntelibX_not_a_number`.

Классы, не относящиеся к трём вышеописанным иерархиям, т.е. не являющиеся ни S-выражением, ни умным либо ведущим указателем S-выражения, ни классом ошибок, именуются с обязательным использованием префикса `Intelib` (для нижнего слоя библиотеки), либо префиксов `Lisp`, `Scheme`, `Refal` для соответствующих подсистем. Так, класс синтаксического анализатора текстового представления S-выражений называется `IntelibReader`, а его потомок, адаптированный под конкретику кода на Scheme, называется `SchemeReader`.

В соответствии с этими соглашениями класс, соответствующий понятию *продолжения*, изначально назывался `SchemeContinuation`. Такого названия мы и будем придерживаться в изложении.

4.2 Основные принципы реализации класса `SchemeContinuation`

Основой реализации класса `SchemeContinuation` являются стек действий (*ToDo stack*) и стек результатов (*result stack*). Стек результатов реализован в виде массива элементов типа `SReference`, что позволяет использовать фрагменты этого стека в качестве векторов параметров при вызове функций, избегая, таким образом, выделения динамической памяти при каждом вызове Scheme-функции. Сам массив создаётся в динамической памяти при создании объекта `SchemeContinuation` и при необходимости расширяется (каждый раз вдвое). Практика показала, что такая необходимость (при начальном размере массива 256 элементов) возникает редко.

Стек действий также представляет собой динамически расширяемый массив, но его элементы имеют более сложную структуру. Каждый элемент содержит код операции из некоторого предопределённого множества, опциональный параметр, а также поле для хранения отладочной информации о стеке вызовов Lisp-форм; последнее может быть исключено при компиляции библиотеки заданием соответствующей директивы условной компиляции.

Кроме этих двух стеков, в объекте хранится также указатель на текущий (активный) «лексический контекст». Класс также вводит два статических поля для поддержки механизма прерывания вычислений.

4.3 Основные коды операций

Работа с объектом `SchemeContinuation` обычно начинается с помещения в стек действий операции «просто вычислить» с параметром, в качестве которого выступает подлежащее вычислению выражение. Соответствующий код операции — `just_evaluate`.

Вторая наиболее часто употребляемая[2] операция — вызов функции для заданного количества аргументов. Поскольку в языке Scheme сам объект вызываемой функции может быть результатом вычисления, причём такое вычисление, вообще говоря, происходит до вычисления параметров функции, соответствующая операция работает следующим образом: если `N` — заданное количество параметров, то от вершины стека отсчитывается `N` позиций, из полученной позиции извлекается объект функции, после чего для этого объекта производится вызов виртуального метода, ответственного за собственно применение функционального объекта к аргументам, причём в качестве вектора аргументов передаётся адрес позиции в стеке, следующей за позицией, из которой извлечён объект-функция. Указатель стека предварительно откатывается на `N`+1 позицию, так что результат выполнения функции будет записан в элемент массива (стека), ранее содержавший объект функции и не задействованный для хранения параметров.

Из соображений эффективности операция «вызов функции для `N` аргументов» не использует поле параметра для хранения числа `N`. Вместо этого за код именно этой операции принимается *любое неотрицательное число*, записанное в поле кода операции, т.е. число `0` соответствует вызову функции без аргументов, число `1` — вызову унарной функции и т.д. Все остальные операции имеют отрицательные коды, в частности, операция `just_evaluate` имеет код `-1`.

Кроме того, для удобства вводятся следующие команды:

- `evaluate_prepared` вычисляет свой параметр; отличается от `just_evaluate` предположением, что все аргументы уже вычислены. Применяется обычно при вычислении функций высших порядков.

- `evaluate_progn` вычисляет последовательность форм, в качестве которой рассматривается параметр; естественно, параметр должен представлять собой список. Эта операция введена для удобства и применяется, например, при выполнении обычных функций, написанных на Scheme, а также и в других случаях, когда язык предполагает выполнение последовательности форм (в предложениях формы `COND`, в теле `LET` и т.п.).

- `quote_parameter` помещает свой параметр в стек результатов. Применяется при вычислении специальных форм, которые в конце своей работыш (в ходе которой могут активно использоваться оба стека) должны возвращать заранее известный результат.

- `drop_result` убирает из стека результатов находящееся на его вершине значение. Применяется в случаях, если результат вычисления очередной формы должен быть проигнорирован; в частности, это делается для всех элементов списка, поданного команде `evaluate_progn`, кроме последнего. Параметр игнорируется.

- `return_unspecified` — помещает в стек результатов значение, соответствующее понятию *unspecified*; это значение, в качестве которого выступает специально созданный объект

[2]Заметим, пример, приведённый в таблице 1, использует только эти две команды

атомарного S-выражения, возвращается из всех функций и форм, относительно которых в стандарте Scheme указано, что возвращаемое значение не определено. Параметр игнорируется.

4.4 Реализация условных конструкций

Реализация специальных форм, осуществляющих вычисления в зависимости от результатов предыдущих вычислений (таких как `IF`, `COND`, `AND` и `OR`) оказывается нетривиальной задачей в силу вышеупомянутой недопустимости рекурсивного вызова вычислителя. Так, при вычислении формы `COND` для каждого предложения необходимо вычислить его первый элемент, и если результатом вычисления будет `#F` (логическая ложь), перейти к следующему предложению, в противном случае продолжить вычисление форм текущего предложения, а остальные предложения пропустить. Однако из кода написанной на C++ функции, ответственной за вычисление формы `COND`, мы *не имеем права произвести вычисление первой формы предложения*, как и любой другой формы. Всё, что можно сделать — это поместить те или иные данные в стеки (как в стек действий, так и, при необходимости, в стек результатов) и вернуть управление.

В процессе реализации оказалось, что введённых выше кодов операций недостаточно; вообще, рассматриваемая реализация базировалась на введении средств *ad hoc*, по мере возникновения конкретных потребностей.

4.4.1 Форма `COND`

Для реализации формы `COND` были введены два кода операций, `cond_clause` и `end_of_clauses`. Первая из них извлекает из стека результатов значение, находящееся на вершине стека (то есть последнее вычисленное). Если значение представляет собой логическую ложь, операция на этом заканчивается; в противном случае параметр операции рассматривается как список форм, подлежащих вычислению (то есть в качестве параметра задаётся остаток `COND`–предложения). Прежде чем приступить к вычислению форм из своего параметра, операция `cond_clause` изымает из стека действий все имеющиеся там действия одно за другим до тех пор, пока не обнаружит в очередном элементе стека команду `end_of_clauses` (таким образом мы избегаем вычисления оставшихся предложений формы `COND`). После этого в стек помещаются команды для последовательного вычисления форм текущего предложения (элементов списка, заданного в параметре операции), причём после каждого, кроме последнего, помещается команда `drop_result`. Случай пустого списка в параметре рассматривается как специальный; в этой ситуации вместо планирования дальнейших вычислений выполняется действие существенно более простое, а именно, значение, только что извлечённое из стека результатов, помещается обратно. Именно оно в этом случае будет конечным результатом вычисления формы.

Команда `end_of_clause` представляет собой «не-операцию» (no-op), то есть, будучи извлечённой из стека действий, не делает ровным счётом ничего. Используется она исключительно для пометки позиции в стеке действий, соответствующей концу формы `COND` или другой формы.

Используя эти два кода операций, мы можем, имея форму `COND`, запланировать её вычисление следующим образом. Сначала поместим в стек действий команду `end_of_clauses` (она, таким образом, окажется глубже всего остального, имеющего отношение к нашей форме). Далее на случай, если ни одно предложение `COND` не будет выполнено (то есть все условия при вычислении дадут значение «ложь») поместим в стек действий команду `quote_parameter` с параметром `#F`.

Далее просмотрим предложения формы `COND` в обратном порядке (в имеющейся реализации это делается на обратном ходе рекурсии) и для каждого предложения запишем в стек действий сначала команду `cond_clause` с «хвостом» предложения в качестве параметра, затем команду `just_evaluate` с головой предложения в качестве параметра. После этого можно будет вернуть управление.

Итак, дальнейшее выполнение будет происходить следующим образом. Будет извлечена из стека команда `just_evaluate`, параметром которой является первый элемент (т.е. усло-

вие) первого предложения COND; после её выполнения (которое может, разумеется, потребовать сколь угодно сложных действий, но все эти действия затронут только позиции обоих стеков, находящиеся выше рассматриваемых) на вершине стека результатов окажется логическое значение, показывающее, следует ли выполнять текущее предложение COND или перейти к следующему. Далее из стека действий будет извлечена команда cond_clause, которая этим значением воспользуется, и так для каждого предложения COND. Если одно из предложений будет выполняться (то есть его условие даст значение истины), весь остаток операций, помещённых в стек формой COND, будет из стека изъят, а затем будут вычислены формы текущего предложения, причём в стеке результатов останется только результат последней из них.

4.4.2 Формы IF, AND и OR

Механизм, подобный вышеописанному, был применён и для реализации других условных форм.

В частности, для формы IF оказалось достаточно уже введённых команд. При её вычислении в стек действий помещается сначала команда end_of_clauses; как и для формы COND, она оказывается в стеке глубже всех. Затем в зависимости от наличия или отсутствия в форме ветки *else* (т.е. третьего аргумента формы) в стек действий помещается либо команда just_evaluate с этим третьим аргументом в качестве параметра, либо команда quote_parameter с параметром «ложь». Затем в стек действий записывается команда cond_clause с параметром, представляющим собою список из одного элемента — ветки *then* (то есть второго параметра формы IF). Наконец, последним в стек записывается команда just_evaluate с параметром, равным первому параметру формы IF., после чего можно вернуть управление. Нетрудно видеть, что дальнейшие вычисления по своей семантике будут в точности соответствовать порядку действий, предполагаемому формой IF.

Форма OR реализуется просто как частный случай формы COND со всеми предложениями, состоящими только из первого элемента. Иначе говоря, в стек действий сначала помещается команда end_of_clauses, затем quote_parameter с параметром #F, а затем в обратном порядке для каждого аргумента формы — команда cond_clause с пустым списком в качестве параметра и команда just_evaluate с параметром, соответствующим рассматриваемому аргументу формы.

Чуть сложнее обстоят дела с формой AND, поскольку здесь следует прекратить дальнейшие вычисления (изъять из стека действий всё до метки end_of_clauses) не по значению «истина», а, наоборот, по значению «ложь». Поэтому для реализации формы AND пришлось ввести ещё одну операцию, которая названа bail_on_false. По своей семантике она аналогична (и в известном смысле противоположна) команде cond_clause: при её выполнении из стека результатов извлекается значение, и если оно истинно, то не делается больше ничего, если же оно ложно, то из стека действий изымаются все команды до метки end_of_clauses, а затем в стек результатов записывается значение «ложь». Команда bail_on_false не использует (игнорирует) свой параметр. Таким образом, для вычисления формы AND в стек действий необходимо поместить end_of_clauses, затем just_evaluate с последним аргументом формы AND в качестве параметра, затем для всех аргументов формы, кроме последнего, в обратном порядке — bail_on_false и just_evaluate с рассматриваемым аргументом формы в качестве параметра.

4.5 Реализация итерационных вычислений

Функции и формы, требующие итерационного исполнения и вычислений форм на каждой итерации, оказываются ещё более нетривиальны в реализации, когда прямой вызов вычислителя запрещён. К этой категории функций относятся, например, стандартные функция MAP и форма DO.

В рассматриваемой реализации для этой цели введены, во-первых, две дополнительные команды generic_iteration и iteration_callback, и, во-вторых, вспомогательный абстрактный класс SExpressionGenericIteration, наследники которого реализуют конкретные итерационные процессы. Класс вводит четыре чисто виртуальных метода: NeedAnotherIteration, ScheduleIteration, CollectResultOfIteration и ReturnFinalValue, которые необходимо

определить в классе-потомке. Метод `NeedAnotherIteration` возвращает булевское значение, показывающее, следует ли выполнять ещё одну итерацию цикла. Предполагается, что этот метод ничего не меняет в объекте и не производит никаких активных действий, а только вычисляет на основании тех или иных признаков, завершен ли итерационный процесс. Задачей метода `ScheduleIteration` является размещение в стеке действий набора команд, необходимых для выполнения очередной итерации; метод `CollectResultOfIteration` вызывается после выполнения итерации, чтобы дать возможность объекту извлечь из стека результатов итоги выполнения очередной итерации. Наконец, метод `ReturnFinalValue` вызывается после того, как `NeedAnotherIteration` вернул логическую ложь. Этот метод должен поместить в стек результатов то значение, которое будет использовано в качестве итогового результата цикла; он также может вместо готового результата поместить в стек действий инструкции по его вычислению.

Команда `generic_iteration` выполняется следующим образом. Из параметра команды извлекается объект класса `SExpressionGenericIteration` и для этого объекта вызывается метод `NeedAnotherIteration`. Если метод вернул ложь, вызывается метод `ReturnFinalValue` и работа на этом заканчивается. Если же была возвращена истина, то в стек действий заносятся сначала команда `generic_iteration` (чтобы описываемые действия были повторены после выполнения очередной итерации), затем команда `iteration_callback` с тем же объектом `GenericInteration` в качестве параметра, и, наконец, вызывается метод `ScheduleIteration` для добавления в стек действий инструкций, необходимых для выполнения очередной итерации.

Команда `iteration_callback` выполняется проще: предполагая, что её параметром является объект `SExpressionGenericIteration`, она вызывает его метод `CollectResultOfIteration`.

Таким образом, для выполнения произвольного итерационного процесса достаточно описать соответствующий класс (потомок `SExpressionGenericIteration`), сконструировать его объект и поместить в стек действий команду `generic_iteration` с этим объектом в качестве параметра.

4.6 Работа с контекстами и присваиваниями

Часто возникает потребность изменить содержимое области памяти, содержащей S-выражение (например, присвоить значение Scheme-переменной), причём новое значение, а иногда и расположение изменяемой памяти является результатом вычислений; такое случается, например, при выполнении формы `SET!`.

Следует отметить, что любое присваивание в терминах Scheme является на самом деле изменением значения объекта класса `SReference`. Для выполнения присваиваний вводится ещё один вспомогательный класс, `SExpressionLocation`, который инкапсулирует указатель на `SReference`. Для присваивания вводятся две команды, `assign_to` и `assign_location`. Обе извлекают значение из стека результатов и заносят его по адресу, заданному объектом `SExpressionLocation`, только `assign_to` берёт этот объект из своего параметра, а `assign_location` — из стека результатов, позволяя, таким образом, производить присваивание в вычисленную позицию.

Отметим, что объект `SExpressionLocation` способен, кроме собственно указателя на позицию в памяти, хранить ещё и «умную ссылку» на структуру данных, *содержащую* эту позицию. Например, если указатель указывает на car или cdr от точечной пары, то «умную ссылку» есть смысл установить на саму эту точечную пару. Это гарантирует, что соответствующая позиция не перестанет быть валидной, пока существует объект `SExpressionLocation`, указывающий на данную позицию.

Вычисление некоторых форм (например, форм семейства `LET`) предполагает манипуляцию контекстами: например, при выполнении обычной формы `LET` вычисление значений, подлежащих связыванию с переменными, производится во внешнем контексте, а для связывания создаётся ещё один (уже внутренний) контекст, в котором и производится связывание, а также и вычисление форм, образующих тело `LET`. Контекст представляется классом `SchExpressionContext`; смена контекста производится командой `set_context`, которая берёт объект контекста из параметра.

Отметим один важный момент. Обычно после вычисления формы необходимо восстановление контекста, бывшего активным до начала её вычисления. Потому при планировании вычисления формы, вводящей локальный контекст, в стек действий заносится операция `set_context` с параметром, равным текущему контексту. Если не предпринять специальных мер, такая техника может привести к расходованию стека в ситуации остаточной рекурсии и, таким образом, свести на нет эффект от оптимизации остаточной рекурсии. В связи с этим функция, отвечающая за занесение в стек действий, при попытке занесения очередной команды `set_context` проверяет, не находится ли на вершине стека другая команда `set_context`, и если это так, не производит занесения новой команды. Это не нарушает семантики вычислений, поскольку эффект от выполнения двух команд `set_context` подряд заведомо эквивалентен эффекту от выполнения последней из них (то есть находящейся глубже в стеке) и, следовательно, новая команда просто не нужна.

4.7 Поддержка моделей возврата

Возврат значений из Scheme-функций, написанных на C++, может, как уже говорилось, осуществляться по одной из нескольких моделей. Для поддержки этих моделей класс `SchemeContinuation` имеет специальные методы:

```
void RegularReturn(const SReference &ref);
void ReferenceReturn(SReference &ref, const SReference &superstruct);
void TailReturn(const SReference &ref);
```

Первый из этих методов, `RegularReturn`, предназначен для обычного возврата значения и, как правило, применяется для значения, которое только что построено. Работает метод очень просто: переданное ему значение он помещает в стек результатов.

Второй метод, `ReferenceReturn`, предназначен для возврата из функций, выбирающих подвыражение из некоего выражения. Таковы, например, функции `CAR` и `CDR`: они возвращают не новое выражение, а *часть существующего выражения*. Поведение метода ReferenceReturn зависит от текущего содержимого стека действий. В случае, если на вершине стека находится операция `assign_location` (или эта операция отделена от вершины стека операцией `set_context`, что тоже допустимо), метод `ReferenceReturn` формирует объект `SExpressionLocation` из своих аргументов. В противном случае метод работает как для простого возврата, то есть помещает свой первый аргумент в стек результатов. Наличие этого метода позволяет реализовать функцию, подобную форме `SETF` из Common Lisp.

Наконец, последний из трёх методов, `TailReturn`, предназначен для оптимизации остаточной рекурсии. Этот метод применяется в случае, если необходимо вернуть в качестве значения результат очередного вычисления. Работает метод очень просто: помещает в стек действий команду `just_evaluate`, а свой аргумент передаёт ей в качестве параметра.

5 Взаимовлияние диалектов InteLib Scheme и InteLib Lisp

После завершения первой рабочей реализации InteLib Scheme было принято решение переписать реализацию диалекта InteLib Lisp с использованием новых механизмов. Это позволило добиться повышения временно́й эффективности работы Lisp-вычислителя в несколько раз; скорее всего, основная экономия достигается за счёт использования фрагментов стека результатов в качестве вектора параметров при вызовах функций.

Перечислим основные принципиальные различия между диалектами InteLib Lisp и InteLib Scheme:

1. В Scheme вычисляются все элементы формы, если только первый элемент не является символом, с которым связана специальная синтаксическая конструкция; в диалекте Lisp первый элемент формы не вычисляется и должен представлять собой либо символ, либо лямбда-список.

2. В Scheme символ имеет только одно значение; функция, ассоцированная с символом, является значением символа. В диалекте Lisp символ имеет, независимо друг от друга, значение и ассоциированную функцию.

3. В диалекте Lisp имеются динамически связываемые символы, вводимые формой `DEFVAR`; в Scheme таких нет.

4. В диалекте Lisp роли пустого списка и логической лжи совмещены и исполняются символом `NIL`. В Scheme пустой список и логическая ложь обозначаются специальными объектами, различающимися между собой и не являющимися символами.

Естественно, реализации вычислительных моделей языков Scheme и Lisp существенно различаются. Прежде всего это относится к алгоритму вычисления S-выражения, а также к обработке «лексических контекстов». Тем не менее, существенное количество кода оказывается для реализации этих моделей общим.

При реализации общая часть кода класса `SchemeContinuation` была выделена в отдельный абстрактный базовый класс, получивший название `IntelibContinuation`. В этом классе описываются два виртуальных метода: `JustEvaluate`, призванный задавать общий алгоритм вычисления выражения (чисто виртуальный) и `CustomCommand` для реализации дополнительных кодов операций (в базовом классе пустой). Кроме того, в классе предусмотрены методы для работы с логическими значениями. Это позволило реализовать ряд библиотечных функций в одном экземпляре для обоих диалектов (так, функция `STRING?` для Scheme и функция `STRINGP` для диалекта Lisp имеют одну и ту же реализацию).

Реализация всех описанных выше кодов операций была вынесена в класс `IntelibContinuation`. В новой версии класса `SchemeContinuation`, наследуемой от `IntelibContinuation`, вводится одна дополнительная команда `case_check` для реализации формы `CASE`. В классе `LispContinuation`, предназначенном для диалекта Lisp, вводятся две дополнительные команды, а именно, `take_result_as_form` (вычисляет форму, находящуюся на вершине стека результатов; используется в реализации макросов) и `duplicate_last_result` (извлекает из стека результатов одно значение и помещает его обратно дважды; используется в реализации `SETQ` и `SETF`).

В общую часть реализации вычислительных моделей Lisp и Scheme были вынесены также базовые классы, представляющие понятия функции и спецформы (спецсинтаксиса в терминах Scheme). Реализация множества библиотечных функций оказалась не зависящей от диалекта; в качестве примера можно назвать функцию `COND`.

С другой стороны, для нужд диалекта Lisp пришлось выполнить реализацию динамического связывания переменных, реализовать концепцию обобщённого присваивания (`SETF`), сделать нерекурсивную реализацию функции `SORT`, аналогичной одноимённой функции из Common Lisp [10], которая предполагает передачу в качестве параметров, во-первых, функции, задающей отношение порядка и, во-вторых, функции-селектора.[3]

После реализации указанных возможностей для диалекта Lisp показалось естественным ввести соответствующие возможности и в Scheme. Таким образом, в настоящий момент InteLib Scheme является единственным диалектом языка Scheme, поддерживающим форму `SETF`; кроме того, поддерживается и функция `SORT`, во многих случаях весьма удобная. С другой стороны, в InteLib Lisp была введена поддержка `CALL/CC`.

6 Заключение

Полученный опыт позволяет сделать несколько неожиданное предположение о том, что вычислительные модели, основанные на рекурсии, следует реализовывать без использования рекурсии. Такое предположение нуждается, безусловно, в дальнейшей экспериментальной проверке и теоретическом обосновании.

Полученный в ходе работы диалект языка Scheme не соответствует существующим стандартам этого языка. С одной стороны, некоторые возможности в имеющемся диалекте не были реализованы за отсутствием достаточного времени (это касается прежде всего подсистемы макросов). С другой стороны, некоторые возможности, предусмотренные стандартом, плохо вписываются в чисто компилируемое окружение, определяемое использованием C++. В частности, к этой категории возможностей относится поиск символа по имени и извлечение

[3] Интересно заметить, что в пятой версии стандарта Scheme [9] функции `SORT` просто нет.

имени символа: действительно, имена переменных, роль которых в Scheme играют символы — это сущность, которая в чисто компилируемом коде должна полностью исчезать во время компиляции (например, результирующий исполняемый код не должен, по идее, изменяться при переименовании переменных в исходном коде).

В то же время, реализация, выполненная в рамках проекта InteLib, может быть использована и для синтеза встроенных интерпретаторов, и в этом случае стандартные возможности, обусловленные интерпретируемой сущностью Scheme, могут оказаться полезны.

Сказанное приводит нас к идее создания иерархии *уровней совместимости* реализуемого диалекта со стандартами в зависимости от конкретной ситуации. Выделение таких уровней и определение набора возможностей, доступных на каждом из них, представляется логичным направлением дальнейших исследований в области интеграции вычислительной модели языка Scheme в проекты на C++.

Список литературы

[1] E. Bolshakova and A. Stolyarov. Building functional techniques into an object-oriented system. In *Knowledge-Based Software Engineering. Proceedings of the 4th JCKBSE*, volume 62 of *Frontiers in Artificial Intelligence and Applications*, pages 101–106, Brno, Czech Republic, September 2000. IOS Press, Amsterdam.

[2] А. В. Столяров. Интеграция изобразительных средств альтернативных языков программирования в проекты на C++. Рукопись депонирована в ВИНИТИ 06.11.2001, №2319-В2001, Москва, 2001.

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

[4] А. В. Столяров. Расширенный функциональный аналог языка Рефал для мультипарадигмального программирования. // Л. Н. Королев, ред., *Программные системы и инструменты. Тематический сборник*, том 2, стр. 184–195. Издательский отдел факультета ВМиК МГУ, Москва, 2001.

[5] А. Столяров, Е. Большакова, Н. Баева. InteLib Lisp в обучении программированию на Лиспе. // Тезисы докладов конференции «Свободное программное обеспечение в высшей школе», Переславль, 28–29 января 2006 года.

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

[7] А. Столяров. Библиотека InteLib — инструмент мультипарадигмального программирования. // II конференция разработчиков свободных программ «На Протве». Тезисы докладов. Обнинск, 25–27 июля 2005 г., стр. 56–62.

[8] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3(4):184–195, Apr 1960.

[9] R. Kelsey, W. Clinger, and J. Rees. Revised^5 report on Algorithmic Language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, 1998.

[10] G. L. Steele Jr. *Common Lisp the Language.* Digital Press, Burlington MA, second edition, 1990.

[11] Dan Piponi. Continuations in C.
`http://homepage.mac.com/sigfpe/Computing/continuations.html`

[12] J. Alger. *C++ for real programmers.* AP Professional, Boston, 1998. Русский перевод: *Джефф Элджер*, C++: библиотека программиста. СПб., Питер, 2001.