УДК 681.3.07

ПРИНЦИПЫ КОМПИЛЯЦИИ ПРОГРАММ НА ЯЗЫКЕ SCHEME С ИСПОЛЬЗОВАНИЕМ МЕТОДА CPS

Д.Ю. Турдаков, М.Н. Гринев

Рассмотрен один из наиболее перспективных подходов к реализации компиляторов для функциональных языков с «энергичной» семантикой, таких как Scheme и ML. В качестве примера использован компилятор Gambit-C для языка Scheme. Приведено краткое описание языка Scheme. Рассмотрены принципы компиляции программ на языке Scheme. Исследована проблема свободных переменных. Проведен анализ возможных решений проблемы остаточных вычислений.

Введение

Современное программное обеспечение постоянно усложняется, и становится очевидна необходимость в улучшении его структуры. Требование надежности программного обеспечения приводит к потребности в высокоуровневых языках программирования, реализация которых берет на себя, в частности, ответственность за такой проблемный аспект разработки программных продуктов как управление памятью. Желание скорейшего выхода на рынок с новыми возможностями программного обеспечения вынуждает искать языки программирования, способные существенно повысить производительность труда рядовых разработчиков. Ответ на вышеперечисленные проблемы многие разработчики находят для себя в использовании функциональных языков программирования.

Среди основных черт функциональных языков, благодаря которым эти языки находят своих приверженцев, можно выделить следующее [1]. Во-первых, это поддержка стиля программирования, основанного на концепции *прозрачности ссылок* (referential transparency). Прозрачность ссылок достигается за счет явной передачи параметров по значению, в отличие от используемой в императивном стиле программирования возможности передачи ссылок на переменные. При этом оптимизационная работа, направленная на избежание избыточного копирования значений, перекладывается с программиста на компилятор, за счет чего удается избежать большого числа ошибок, появляющихся в результате многочисленных побочных эффектов при работе со ссылками. Во-вторых, функциональные языки позволяют трактовать функции как значения и передавать их в качестве параметров наравне со значениями других типов. Возможность работать с функциями как со значениями позволяет улучшить модульность разрабатываемой системы. В-третьих, в основу большинства функциональных языков положена формальная модель лямбда-исчисления, что дает возможность использовать формальные методы изучения свойств функциональных программ. Помимо трех перечисленных черт существует и ряд других, которыми обладают некоторые функциональные языки, например, поддержка средств сопоставления по образцу (pattern matching) или возможность статического вывода типов для проверки корректности программы (static type checking).

Кроме того, ряд функциональных языков (такие как Haskell [2]) основан на «ленивой» (lazy) семантике (в отличие от «энергичной» - strict - семантики других языков), что позволяет повысить уровень декларативности языка по сравнению с энергичными языками.

На ранних этапах развития функциональных языков платой за возможность использования перечисленных высокоуровневых средств было снижение производительности результирующих программных систем. Однако современные реализации энергичных функциональных языков успешно конкурируют по производительности с реализациями императивных языков [3]. К сожалению, реализации ленивых функциональных языков остаются по-прежнему достаточно медленными на многих практически важных тестах.

В этой статье мы рассмотрим один из наиболее популярных сегодня подходов к реализации компиляторов для энергичных функциональных языков на примере компилятора Gambit-C для языка Scheme (одного из наиболее популярных диалектов языка LISP). Gambit-C разрабатывается группой Марка Фили (Marc Feeley), профессора Монреальского университета в Канаде с 1990 года. Оптимизирующий компилятор Gambit-C признан одной из наиболее качественных реализа-

ций языка Scheme и, используется во многих практических проектах. В данной статье описание реализации языка написано по мотивам лекции Марка Фили «Реализация Scheme за 90 минут» [4]. Материалов этой лекции и, соответственно, нашей статьи достаточно для написания собственного компилятора, поддерживающего большинство основных возможностей языка Scheme. В [4] можно найти исходный код такого упрощенного компилятора.

Прежде чем перейти к основной теме статьи, требуется пояснить, почему мы считаем, что материал этой статьи может оказать полезным российским разработчикам программного обеспечения. Благодаря своему высокому уровню функциональные языки очень часто используются как языки расширений, встраиваемых в приложения. Чаще всего для этих целей использовались диалекты языка LISP (в частности, Scheme), например, в таких широко известных приложениях как AutoCAD, emacs и GIMP. Российские компании, производящие собственные программные продукты, могут быть заинтересованы в предоставлении возможности расширения своих систем путем написания расширений на Scheme. Для эффективной интеграции функционального языка с программным продуктом могут потребоваться углубленные знания принципов реализации этого языка. В частности, авторы этой статьи познакомились с принципами реализации языка Scheme в процессе его интеграции с разрабатываемой в Институте системного программирования РАН системы управления XML-данными Sedna [5] с целью обеспечения сторонним разработчикам возможностей расширения функциональности этой системы. В контексте XML язык Scheme был выбран еще и потому, что, благодаря естественности отображения XML на списочные структуры языка Scheme, на его основе удалось разработать очень удобные средства обработки XML-данных, опирающиеся на подход SXML [6].

2. Введение в Scheme

В этом разделе приводится краткое описание языка Scheme. Информации этого раздела достаточно для понимания материалов всей статьи. Полную информацию о языке Scheme можно найти в спецификации языка [7].

2.1. Базовые принципы языка Scheme

Одной из ключевых черт языка Scheme является возможность работы с функциями как со значениями: их можно создавать динамически в процессе выполнения программы, присваивать переменным, сохранять как элементы в структурных типах данных, передавать как значения параметров в функции и возвращать как результат вычисления выражения. Про уравнивание функций в правах с другими значениями говорят, что это делает их «значениями первого класса».

В языке Scheme значения (такие как числа, строки, векторы, функции) неявно характеризуются местом, где они расположены - *местоположением* (location), или последовательностью таких мест. Например, строка характеризуется последовательностью местоположений каждого символа этой строки. Для простоты восприятия под местоположением можно понимать некоторый адрес оперативной памяти, однако заметим, что в стандарте языка понятие местоположения не уточняется. Язык Scheme предоставляет средства доступа к значениям и их модификации по их местоположению. Например, стандартная функция string-set! позволяет сохранить новое значение по определенному местоположению в строке (местоположение задается его порядковым номером). *Переменной* в языке Scheme называется идентификатор, именующий некоторое местоположение. Говорят, что переменная *связана* с местоположением. Таким образом, переменная языка Scheme связана с местоположением, а не с тем значением, которые находится по этому местоположению. Переменная может быть использована для изменения значения, находящегося по соответствующему местоположению, при помощи оператора присваивания set!. Фактически, введение понятия местоположения и соответствующих операций изменения значений по местоположению, выводит Scheme за рамки чисто функционального языка и допускает императивный стиль программирования. Тем не менее, если придерживаться чисто функционального стиля программирования, можно забыть про понятие местоположения и считать, что переменная связана непосредственно со значением.

Множество связываний, которые видны в некоторой точке программы, называется *активным окружением* (environment in effect). В языке Scheme поддерживается ряд выражений, позволяющих расширять активное окружение новыми связываниями путем создания *новых* местоположений и связывания переменных с созданными местоположениями. Такие выражения называются

связывающими (binging constructs). Фундаментальным связывающим выражением является *лямбда-выражение* (выражение, позволяющее создать новую функцию); все остальные связывающие выражения вводятся как макросы, определяемые через лямбда-выражение.

Как и большинство языков программирования, язык Scheme имеет *блочную структуру*. То есть связывания, добавляемые связывающим выражением, расширяют активное окружение только для тела связывающего выражения. При этом добавляемые связывания перекрывают связывания одноименных переменных, которые были добавлены внешними связывающими выражениями. В последующих разделах содержится большое количество примеров использования лямбда-выражений для создания новых связываний, расширяющих активное окружение.

Разрешение ссылки на переменную (то есть выбор соответствующего связывания для получения доступа к значению переменной) осуществляется в языке Scheme в статике с учетом блочной структуры программы. Ко времени выполнения программы все ссылки на переменные заменяются ссылками на соответствующие местоположения. Как будет показано ниже, реализация статического связывания является нетривиальной задачей вследствие того, что в теле лямбда-выражения могут содержаться ссылки на переменные связанные во внешних лямбда-выражениях.

Язык Scheme является динамически (или слабо) типизированным языком. При динамической типизации, в отличие от статической (или сильной) типизации, тип ассоциируется со значением, а не с переменной.

Другой важной особенностью языка Scheme является то, что аргументы передаются в функции всегда только по значению. Это соответствует *энергичной семантики* вычисления и означает, что выражения, определяющие значения аргументов, вычисляются до начала вычисления функции вне зависимости от того, требуется ли этот аргумент для вычисления значения функции. Энергичная семантика вычислений языка Scheme отличается от *ленивой семантики* языка Haskell и от *семантики передачи значения по имени* языка Algol 60. В этих языках выражение, определяющее значение аргумента, не вычисляется до тех пор, пока это значение не будет востребовано при вычислении функции.

В соответствии со спецификацией, реализация языка Scheme обязана выполнять *хвостовые вызовы* без использования дополнительной памяти. Эффективная обработка хвостовых вызовов имеет очень большое значение, поскольку при написании программ в функциональном стиле итерационные вычисления, выражаемые в императивных языках через циклические операторы, записываются через рекурсивные функции. В том случае, если рекурсивные вызовы являются хвостовыми, разработчик может не опасаться за выход за пределы доступной приложению памяти. Хвостовые вызовы и их эффективное выполнение подробно рассматриваются в нашей статье ниже.

Помимо функций, в языке Scheme статус значений первого класса также получили остаточные вычисления (continuations), обработка которых в большинстве языков программирования осуществляется неявно. Механизм остаточных вычислений позволяет реализовывать многие управляющие конструкции, например, обработку исключительных ситуаций. Далее в этой статье мы будем уделять особое внимание остаточным вычислениям и принципам их поддержке в языке Scheme, поскольку эффективная поддержка механизма остаточных вычислений является одной из наиболее сложных задач реализации Scheme.

2.2. Синтаксис

Для записи как программ, так и данных в языке Scheme используется префиксная нотация, в которой каждое подвыражение обязательно должно быть заключено в скобки. Такой синтаксис отличает Scheme от синтаксиса многих других языков. Например, в языке C используется инфиксная запись с возможностью опускать скобки. Например, выражение, которое на языке C записывается как *2+3*4,* на языке Scheme будет выглядеть следующим образом:

```
( + 2  (*  3  4))
```

В зависимости от контекста это выражение может интерпретироваться как литеральная константа или как программа. При интерпретации в качестве программы предполагается, что первое выражение после открывающей скобки возвращает функцию, для которой значения всех остальных выражений выступают в роли аргументов. Так, в нашем примере выражение + является ссылкой на переменную, связанную с функцией сложения. Эта функция будет применена к ре-

зультатам выражений 2 и (* 3 4). У такой нотации имеется несколько преимуществ. Во-первых, не существует различия между применением операции и вызовом функции. Фактически, операции в Scheme - это обычные функции, что позволяет делать их не только унарными и бинарными. Во-вторых, поскольку всегда явно расставляются скобки, нет необходимости в поддержке уровней приоритетов операций, в противовес 15 уровням приоритета в языке C.

2.3. Выражения

Выражения языка Scheme разделяются на *базовые* и *производные*. В спецификации определено 6 видов базовых выражений: ссылки на переменные, литеральные выражения, функции, вызовы функций, условные выражения и присваивания. На основе базовых выражений представляются все производные выражения. В этой статье мы подробно рассмотрим только базовые выражения. Определения производных выражений можно найти в спецификации языка Scheme [7].

2.3.1. Ссылка на переменную

Выражение, состоящее из имени переменной, называется *ссылкой на переменную*. Результатом такого выражения является значение, находящееся по местоположению, с которым связана данная переменная в активном окружении. Ссылка на несвязанную переменную является ошибкой.

2.3.2. Литеральные выражения

Литеральные выражения используются для включения в текст программы константных значений. Литеральные выражения записываются по следующим правилам:

(quote <*datum*>)
'<datum>
<constant>

Выражение (quote <*datum*>) вычисляется в <*datum*>. <datum> может быть любым допустимым *внешним представлением* произвольного значения языка Scheme. Правила записи внешних представлений определены в языке Scheme для большинства типов значений за исключением значений, представляющих функции. Например, значение целого типа имеет внешнее представление в виде последовательности символов "28"; список из трех целых числе 1, 2, 3 имеет внешнее представление «(1 2 3)». Выражение '<datum> является сокращенной формой от (quote <*datum*>) . Для числовых, строковых, символьных и логических констант <constant> не требуется использование quote, например, выражения "abc" и (quote "abc") эквивалентны.

2.3.3. Функции и вызовы функций

Как говорилось выше, функции в Scheme являются значениями первого класса. Это означает, что они могут выступать наравне со значениями других видов в качестве аргументов функций, в качестве результата значения функций, а также могут создаваться динамически при вычислении программы.

Создать функцию можно с помощью лямбда-выражения, которое подчиняется следующим синтаксическим правилам:

(lambda (<формальные-параметры>) <выражение-тело>)

Слово «lambda» происходит из лямбда-исчисления, которое является теоретической основой языка Scheme. Результатом вычисления лямбда-выражения является безымянная функция. Например, выражение ***(lambda (x) (* x x))*** возвращает функцию, вычисляющую квадрат значения своего единственного параметра.

Кроме того, как отмечалось выше, ссылки на переменные должны разрешаться в статике, и это требование оказывает сильное влияние на правила вычисления лямбда-выражений. Рассмотрим этот аспект вычисления лямбда-выражений более подробно.

Дело в том, что лямбда-выражение может содержать ссылки на переменные, связанные вне самого лямбда-выражения. Будем называть такие ссылки на переменные *свободными* (по отношению к лямбда-выражению). Например, в приводимом на рис. 1 примере в лямбда-выражении ***(lambda (x) (+ x n))*** имеется свободная ссылка на переменную о. В примере используется выводимая конструкция *let,* которая позволяет расширить активное окружение новыми связыва-

ниями, то есть определить локальные переменные. Первым параметром задается список пар (идентификатор значение), определяющий локальные переменные, вторым параметром является последовательность выражений, для которых будут определены эти локальные переменные.

```
(let ((n 1))
        (let ((f (lambda (x) (+ x n))))
                (let ((n 10))
                        (f 1))))
```

Рис. 1. Лямбда-выражение со свободной ссылкой на переменную

Теоретически можно выделить два подхода к разрешению ссылки на переменную: в статике (как принято в языке Scheme) и в динамике (как принято во многих других диалектах языка LISP). При этом результаты программы будут различные. При статическом разрешении результатом выражения, приведенного на рис. 1, будет 2, а при динамическом разрешении - 11. По сравнению со статическим разрешением, использование динамического связывания позволяет выражать многие сложные вычисления в более компактной форме, но потенциально приводит к большему числу ошибок и усложняет отладку программы [8]. В связи с этим в языке Scheme поддерживается статическое разрешение ссылок на переменные.

Вызов функции записывается в виде заключенного в круглые скобки выражения, возвращающего функцию, и аргументов этой функции!

(<выражение-функция> <аргумент1> ...)

Функции, которые принимают другие функции в качестве аргумента, называются *функциями высокого порядка*. Например, встроенная в Scheme функция *map,* принимает два аргумента: первый аргумент - функция с одним параметром, второй аргумент - список. Функции *map* возвращает вновь построенный список, полученный путем применения к каждому элементу исходного списка функции, переданной в качестве первого аргумента. В следующем примере параметрами вызова функции *map* является безымянная функция, умножающая свой аргумент на два, и список целых чисел. В результате вычисления получаем список соответствующих чисел, умноженных на два:

```
(map (lambda (x) (* x 2)) '(1 2 3 4 5))
=> (2 4 6 8 10)
```

2.3.4. Условные выражения

Условные выражения записываются в следующей форме:

(if <*test*> <*consequent*> <*altemate*>)

В отличие от императивных языков условное выражения в языке Scheme возвращает некоторое значение, которое вычисляется по следующему правилу: если результатом вычисления выражения <*test*> является «истина», то результатом уловного выражения будет результат вычисления выражения <*consequent*>, в противном случае - результат вычисления <*alternate*>.

2.3.5. Присваивание

Как уже отмечалось, язык Scheme не является чисто функциональным языком; в нем поддерживается императивный стиль программирования, основанный на побочных эффектах, которые изменяют значения переменных в ходе выполнения программы. Для изменения значения переменной используется выражение присваивания следующего вида:

(set! <имя переменной> <выражение>)

Результат вычисления <*выражения*> сохраняется по местоположению, с которым связана данная переменная. Результат вычисления выражения присваивания считается неопределенным и зависит от реализации.

2.4. Встроенные типы данных и функции

В языке Scheme поддерживается ряд встроенных типов данных, таких как числовые типы, строковый тип, символьный тип, структурные типы (пара (pair), список и вектор) и некоторые другие, а также встроенные функции для работы со значениями этих типов. Отметим, что любая программа выполняется в активном окружении, которое содержит связывания имен всех встроенных функций с местоположениями, содержащими реализацию этих функций. Таким образом, в качестве имен встроенных функций используются обычные ссылки на переменные.

Подробное рассмотрение всех встроенных типов и функций выходит за рамки нашей статьи и может быть найдено в спецификации языка Scheme [7]. Кратко рассмотрим только две функции

работы с векторами, поскольку они будут использоваться ниже в примерах нашей статьи. Вектор языка Scheme является аналогом массива в языке C. Функция *vector* принимает на вход неограниченное число аргументов и возвращает вновь созданный вектор, содержащий значения этих аргументов в качестве своих элементов. Функция *vector-ref позволяет* вернуть элемент вектора по его порядковому номеру (отсчет ведется с 0). Например, результатом вычисления приведенного ниже выражения будет 3:

```
(let ((m (vector 1 2 3 4 5)))
(vector-ref m 2))
```

2.5. Структура программы

Программа языка Scheme состоит из *последовательностей выражений, определений переменных* и *синтаксических определений.* Выражения были определены в разделе 2.3. Синтаксические определения используются для определения макросов, и не будут рассматриваться в нашей статье. Определения переменных расширяют активное окружение следующих в последовательности за определением выражений новыми связываниями. Отметим, что определения переменных могут появляться не только на глобальном уровне, но и в начале некоторых выражений, например, в начале тела лямбда-выражения. Определения переменных были введены для удобства программирования в императивном стеле, чтобы избежать излишней вложенности, например, при использовании лямбда- и let-выражений. Определение переменной записывается в следующей форме:

```
(define <variable> <expression>)
```

3. Основные принципы компиляции программ на языке Scheme

Перейдем к основной цели этой статьи и покажем процесс трансляции программы, написанной на языке Scheme, в программу на языке C. Практически все компиляторы языка Scheme транслируют Scheme-программы в язык C, а не в машинный код. Это обусловлено тем, что компиляторы языка C существуют для большинства платформ, что обеспечивает переносимость приложений на языке Scheme. При трансляции Scheme-программы на язык C приходится преодолевать ряд трудностей, которые вызваны различиями языков. В отличие от языка C, в Scheme определены следующие понятия и требования, которые мы рассмотрим в дальнейшем подробно:

функции высокого порядка и коррелирующая с ними проблема связывания переменных;

требование эффективно выполнять хвостовые вызовы и связанная с этим проблема роста остаточных вычислений;

возможность работать с остаточными вычислениями как с объектами первого класса, что усложняет предыдущую проблему;

автоматическое управление памятью и сборка мусора.

Очевидно, что основные проблемы при трансляции программ на языке Scheme в программы на языке C возникают как раз в связи с этими различиями, поэтому мы уделим им особое внимание (кроме автоматического управления памятью, описание которого не является целью данной статьи). Поставим перед собой цель преодолеть эти проблемы с помощью средств самого Scheme, переписывая исходные программы с использованием подмножества языка Scheme таким образом, что результирующие программы легко отображаются в C-программу.

В следующих двух разделах статьи мы описываем решения двух ключевых проблем компиляции Scheme-программ: свободных переменных и остаточных вычислений. В результате мы получаем набор правил, преобразующих произвольную Scheme-программу к виду, который может быть легко оттранслирован в C. В заключительном разделе рассматривается генерация кода на примере одного из самых быстрых компиляторов языка Scheme "Gambit".

4. Проблема **свободных переменных и замыкания**

Как уже говорилось, в Scheme используется только статическое связывание. При реализации такого связывания возникает проблема доступа к свободным переменным. Эта проблема появляется из-за наличия функций высокого порядка. Для того чтобы понять, откуда возникают сложности, рассмотрим пример (рис. 2).

Функция *add* с одним параметром *x* возвращает новую функцию, которая прибавляет к своему параметру *y* значение параметра *x* функции *add*. Из примера видно, что параметр *x* используется после того, как функция *add* завершает свою работу. Отсюда становится очевидной неприменимость стековой модели вычислений, и возникает необходимость хранить значения свободных переменных каждой функции до тех пор, пока возможен вызов этой функции. Проблема решается помещением значений свободных переменных и кода функции в один объект.

```
(define add
(lambda (x)
(lambda (y) (+ x y))))
((add 1) 2)          ; результат работы: 3
```

Рис. 2. Свободные переменные и функции высокого порядка

Для обсуждения подходов к решению проблемы будем использовать пример, приведенный на рис. 3.

Первое решение, которое приходит в голову, состоит в увеличении числа параметров и явной передаче значений свободных переменных в качестве аргументов (рис. 4). Это преобразование называется *lambda lifting*.

```
(lambda (x y)
  (let ((f (lambda (a b)
                  (+ (* a x) (* b y)))))
    (– (f 1 2) (f 3 4))))
```

Рис. 3. Проблема свободных переменных

Однако такое решение не идеально, так как будет работать не всегда. Это показывает пример, приведенный на рис. 5.

```
(lambda (x y)
  (let ((f (lambda (x y a b)
                  (+ (* a x) (* b y)))))
    (– (f x y 1 2) (f x y 3 4))))
```

Рис. 4. Преобразование Lambda lifting

```
(define func (lambda (x y)
  (let ((f (lambda (a b)
                  (+ (* a x) (* b y)))))
    f)))
((func 1 2) 3 4) ; результат работы: 11
```

Рис. 5. Случай, в котором lambda lifting не работает

Функция func возвращает в результате своей работы функцию f, к которой используемое преобразование не применимо, так как чтобы вернуть функцию, а не результат ее работы, необходимо записать только имя этой функции без параметров.
Поэтому рассмотрим более универсальный способ.

4.1. Замыкания

Построим структуру, в которой будет храниться сама функция, а следом за ней будут лежать значения ее свободных переменных. Такая структура называется *замыканием*.

Чтобы получить доступ к свободным переменным из тела функции передадим ей на вход полученную структуру (в примере это параметр *self)*. А чтобы вызвать функцию, возьмем ее из структуры и передадим ей все необходимые параметры (рис. 7).

Тело функции
(lambda (self ...

Значение свободной переменной ***x***

Значение свободной переменной ***y***

Рис. 6. Структура замыкания

```
(lambda (x y)
(let ((f (vector
(lambda (self a b)
(+ (* a (vector-ref self 1))
(* b (vector-ref self 2))))
x y)))
(– ((vector-ref f 0) f 1 2)
((vector-ref f 0) f 3 4))))
```

Рис. 7. Явное использование замыкания

Конечно, писать программы в таком виде не удобно. Поэтому возникает необходимость в автоматическом преобразовании написанной программы к такому виду. С помощью простых правил, показанных на рис. 8, можно преобразовать любую программу к требуемой форме (в прямоугольники заключены части, которые необходимо преобразовать).

В прямоугольниках находится часть правила, которую необходимо преобразовывать. Например, (lambda (x) (+ x 1)) преобразуется по первому правилу к виду (vector (lambda (self x) (+ x

1))), при этом выражение (+ x 1) должно быть преобразовано на следующем шаге (с помощью третьего правила).

$\boxed{(lambda\ (P1 \ldots Pn)\ E)} = (vector\ (lambda\ (self\ P_1 \ldots P_n)\ \boxed{E}\ )\ \boxed{v}\ \ldots)$, где v … – список свободных переменных функции $(lambda\ (P_1 \ldots P_n)\ E)$.

$\boxed{v} = (vector\text{-}ref\ self\ i)$, где v свободная переменная, а i – позиция v в списке свободных переменных незамкнутого lambda-выражения

$\boxed{(f\ E_1 \ldots E_n)} = ((vector\text{-}ref\ \boxed{f}\ 0)\ \boxed{f}\ \boxed{E_1} \ldots \boxed{E_n}\ )$.

Рис. 8. Правила преобразования программы для автоматического использования замыканий

5. Проблема остаточных вычислений

5.1. Хвостовые вызовы

В императивных языках циклы основываются на побочных эффектах, которые не одобряются в функциональном стиле программирования. Хотя, благодаря включению в язык операции присваивания, в языке Scheme допускается написание таких циклов, хорошим тоном считается реализация циклов с помощью рекурсии. Требование поддержки хвостовых вызовов, позволяет выполнять рекурсивные программы в Scheme эффективно, разрешая использовать рекурсию любой глубины.

Известно, что эквивалентные рекурсивные программы, можно написать по-разному. На рис. 9 представлены два варианта вычисления факториала числа. В первом случае на каждом шаге рекурсии наблюдается рост *остаточных вычислений* (операции, вычисление которых необходимо продолжить после возврата из текущей функции). Более того, при реализации необходимо где-то сохранять эти остаточные вычисления, чтобы потом вернуться к ним. Во втором случае размер остаточных вычислений ограничен константой, что позволяет эффективно использовать память, так как на каждом шаге рекурсии имеются все необходимые данные.

```
(define fact (lambda (n)
 (if (zero? n) 1 ( * n (fact (- n 1))))))

(fact 4)
=(* 4 (fact 3))
=(* 4 (* 3 (fact 2)))
=(* 4 (* 3 (* 2 (fact 1))))
=(* 4 (* 3 (* 2 (* 1 (fact 0)))))
=(* 4 (* 3 (* 2 (* 1 1))))
=(* 4 (* 3 (* 2 1)))
=(* 4 (* 3 2))
=(* 4 6)
=24
```

```
(define fact-iter (lambda (n)
(fact-iter-acc n 1)))

(define fact-iter-acc (lambda n a)
(if (zero? n) a
(fact-iter-acc (- n 1) (* n a)))))

(fact-iter 4)
=(fact-iter-acc 4 1)
=(fact-iter-acc 3 4)
=(fact-iter-acc 2 12)
=(fact-iter-acc 1 24)
=(fact-iter-acc 0 24)
```

Рис. 9. Два способа вычисления *n!*

Обсудим реализацию этих методов. В первом случае необходимо хранить точки возврата и передаваемые параметры, чтобы впоследствии вернуться на предыдущий шаг и вычислить оставшиеся операции функции. Во втором случае можно реализовать работу с памятью так, чтобы на каждом шаге хранились только необходимые параметры. Возвращаться же не придется, так как вычислять больше нечего, а все необходимые данные у нас есть, то есть отпадает необходимость хранения точек возврата.

Очевидно, что занимаемая память растет с ростом остаточных вычислений, и чем больше этот рост, тем больше используется памяти. Мы увидели, как можно реализовать программу без такого роста. Возникает вопрос, можно ли отобразить любую рекурсивную программу в эквивалентную, но без роста остаточных вычислений.

Для ответа на этот вопрос введем понятие *хвостового вызова*. Интуитивно можно дать такое определение: вызов называется хвостовым, если результатом функции, в которой произошел вы-

зов, будет являться значение, возвращаемое вызванной функцией. Строгое определение задается синтаксически [7].

Теперь определим критерий появления такого роста. Возьмем две любые функции, одна из которых вызывает другую, причем этот вызов хвостовой. Очевидно, что при этом вызове никаких новых остаточных вычислений не добавится, и у всех хвостовых вызовов будут такие же остаточные вычисления, как и у вызывающей функции. Если же вызов не был хвостовым, то необходимо вернуться и вычислить оставшиеся операции. Отсюда становится понятно, что **остаточные вычисления не растут тогда и только тогда, когда все вызовы хвостовые.**

5.2. Остаточные вычисления как объекты первого класса

Одной из самых интересных особенностей языка Scheme является возможность работать с остаточными вычислениями как с объектами первого класса. Разберемся, что это означает, и какие средства для этого предоставляет язык.

Для начала необходимо понять, что же такое «остаточные вычисления». Остаточные вычисления - это «что-то, что ждет значения», чтобы выполнить с ним вычисления. Конечно, это очень расплывчатое определение, но, тем не менее, оно дает возможность представить положение дел. В ходе работы программы с каждым промежуточным значением связано остаточное вычисление, представляющее собой вычисления, которые должны быть выполнены, как только значение станет известным. Остаточные вычисления **не являются** чем-то похожим на функцию, которая получает на вход одно значение и возвращает другое: они получают на вход значение, делают все что следует, и никогда не производят возврат.

Представим выражение *(* (+2 4) (+1 6))*. Здесь можно выделить несколько остаточных вычислений. Остаточные вычисления для *(+2 4)* означают: «взять это значение, и запомнить; сложить 1и 6, взять результат, перемножить его с запомненным значением; остановиться». Остаточные вычисления для *(+ 1 6)* означают: «взять это значение, перемножить его со значением, которое было вычислено ранее (6), и остановиться». Заметим, что результат выражения *(+2 4)* является частью остаточных вычислений *(+ 1 6),* потому что он был вычислен и запомнен ранее.

Остаточные вычисления не являются статическим объектом, который можно определить во время компиляции, они представляют собой динамические сущности, создаваемые и вызываемые во время выполнения программы. На каждом шаге программы, когда вычисляется значение, существуют *текущие* (current) остаточные вычисления, ожидающие значения. Они продолжат обрабатывать операции, которые остается выполнить, включая вычисление других значений и вызовы других остаточных вычислений.

Если рассмотреть остаточные вычисления в контексте стековой модели организации вычислений, то в каждый момент времени они представляют собой стек выполнения, то есть последовательность вложенных функций, которым будет дано на вход значение.

Для работы с остаточными вычислениями в языке Scheme используется функция call/cc (call with current continuation).

```
(call/cc (lambda (cont) <body>) )
```

Функция *call/cc* принимает на вход один аргумент, который, в свою очередь, также должен являться функцией с одним параметром. Этот параметр связывается с текущими остаточными вычислениями. Рассмотрим пример:

```
(call/cc (lambda (cont)
           (cont 128)))
```

Параметр *cont* связывается функцией *call/cc* с текущими, на момент времени 1, остаточными вычислениями. Когда, в момент времени 2, происходит обращение к параметру *(cont 128),* остаточным вычислениям (момента времени 1) передается значение 128, поэтому результатом программы будет значение 128. Таким образом, если остаточным вычислениям передается некоторое значение, то можно считать, что функция *call/cc* возвращает это значение.

Обсудим следующий пример:

```
(call/cc (lambda (cont)
           (+ (cont 128) 256)))
```

Здесь функция *call/cc* применяет значение 128 к сохраненным остаточным вычислениям и пытается проделать после этого некоторые операции. Однако ничего такого не случится, так как, как только произойдет обращение к сохраненным остаточным вычислениям *(cont 128),* программа будет считать их текущими и передаст им значение 128, а остаточные вычисления, которые ожидали значение *x,* чтобы вычислить *(+ x 256)* потеряются (их удалит сборщик мусора). Таим образом, результатом программы останется значение 128.

Теперь представим такой случай:

```
(call/cc (lambda (cont) 128)
```

Так как в теле функции не было обращений к сохраненным остаточным вычислениям, эволюция программы будет проходить естественным путем, и функция просто вернет значение 128.

В следующем примере показывается, как можно сохранить текущие остаточные вычисления, и вызывать их несколько раз.

```
(define saved-cont #f) ; описываем новую переменную, ее значение не важно
 (display (call/cc (lambda (cont)
(set! saved-cont cont) 8)))
(saved-cont 16)
(saved-cont 32)
```

В момент вызова *call/cc* текущие остаточные вычисления имеют вид *(display x).* С ними связывается идентификатор *cont.* После этого в теле функции происходит связывание *(set! saved-cont cont)* сохраненных остаточных вычислений с внешней переменной *saved-cont,* и функция возвращает значение 8, которое в свою очередь будет выведено на экран. Теперь, идентификатор *saved-cont* связан с остаточными вычислениями *(display x),* и, передавая им значения, можно вывести эти значения на экран. Результатом программы будет вывод на экран трех значений: 8, 16, 32.

```
(define (map-/ lst)
(call/cc
(lambda (return)
(map (lambda (x)
(if (= x 0)
(return #f)
(/ 1 x)))
lst))))
(map-/ '(1 2 3)) => (1 1/2 1/3)
(map-/ '(1 0 3)) => #f
```

Рис. 10. Обработка исключительных ситуаций с использованием *call/cc*

Приведем более содержательный пример использования понятия остаточных вычислений. С помощью *call/cc* легко реализуется обработка исключений (рис. 10).

Функция *map-/* выдает список обратных, для каждого элемента входного списка, значений. Если же во входном списке встречается 0, результатом работы будет являться значение *#f.*

5.3. CPS-преобразование

Итак, мы показали, что, во-первых, рост остаточных вычислений приводит к неэффективному использованию памяти, во-вторых, наличие функции для работы с ними *{call/cc)* приводит к тому, что остаточные вычисления могут сохраняться в памяти, существовать неограниченное время и вызываться более одного раза. Отсюда становится очевидна необходимость преобразования программы к виду, при котором все вызовы будут хвостовыми, и память при этом будет использоваться наиболее эффективно. Самым удачным решением оказалось выделение остаточных вычислений в отдельные функции. Этим функциям будут соответствовать объекты, непосредственно управляемые из программы (замыкания), с которыми может легко работать сборщик мусора. Рассмотрим это преобразование на примере (рис. 11).

Остаточные вычисления для *(square 10)* можно представить в виде функции с одним параметром, через который будет передан результат возведения в квадрат: *(lambda (r) (write (+ r 1))).* Так как получившуюся функцию будет необходимо вызывать из тела функции возведения в квадрат, передадим ее как параметр, к которому необходимо обратится после окончания работы *(square 10)* (рис. 12).

```
(let ((square (lambda (x) (* x x))))
      (write (+ (square 10) 1)))
```

Рис. 11. Пример функции для выделения остаточных вычислений

```
(let ((square (lambda (k x) (k (* x x)))))
      (square (lambda (r) (write (+ r 1))) 10))
```

Рис. 12. Функция с выделенными остаточными вычислениями

В общем случае, для осуществления преобразования необходимо для каждой функции проделать три следующих шага:

Представить остаточные вычисления в виде функций с одним параметром.

В заголовок исходной функции добавить еще один параметр, через который будут передаваться эти остаточные вычисления (*continuation*-параметр).

Использовать вызов остаточных вычислений вместо того, чтобы возвращать результат.

Такой способ написания программ называется *Continuation-Passing Style (CPS).*

Чтобы понять суть CPS-преобразования, рассмотрим пример.

```
(let ((mult (lambda (a b) (* a b))))
     (let ((square (lambda (x) (mult x x))))
          (write (+ (square 10) 1))))
```

После преобразования получим

```
(let ((mult (lambda (k a b) (k (* a b)))))
     (let ((square (lambda (k x) (mult k x x))
          (square (lambda (r) (write (+ r 1)))
               10)))
```

Вызов *mult* в *square* - хвостовой, поэтому *mult* имеет те же остаточные вычисления, что и *square.* А все вызовы не являющиеся хвостовыми через *continuation-параметр* получают свои остаточные вычисления.

Таким образом, после CPS-преобразования все вызовы приобретают хвостовую форму и могут быть легко представлены в виде переходов *(jumps)* с текущим набором параметров, что позволяет не хранить в стеке (при стековой модели вычислений) точку возврата и предыдущий контекст.

С помощью CPS-преобразования достигается решение поставленной задачи: остаточные вычисления не растут, память используется наиболее эффективно, а при вызове *call/cc* сохраняется минимальное количество информации.

5.4. Правила CPS-преобразования

Как и с преобразованиями замыканий, можно придумать систему правил для формализации CPS-преобразований, которая позволит автоматизировать весь процесс перевода программы в CPS.

Введем обозначения: $\boxed{\begin{matrix}E\\C\end{matrix}}$ –CPS-преобразование выражения E, где C – остаточные вычисления E. E - исходное выражение (может содержать нехвостовые вызовы). C - выражение в CPS-форме (содержит только хвостовые вызовы), которое представлено либо в виде переменной, либо в виде lambda-выражения.

Первое правило

```
program  =  program
           (lambda (r) (halt r))
```

означает, что самым первым остаточным вычислением программы, является функция, которая получает на вход результат программы и вызывает операцию *(halt r),* завершающую вычисления. Остальные правила показаны на рис. 13а—13в.

С помощью этих правил можно перевести любую программу в CPS. Но что же происходит с *call/cc?*

Если в программе встречается вызов *call/cc,* то он заменяется вызовом *cps-call/cc,* а в программу вставляется определение функции *cps-call/cc.*

```
(define (cps-call/cc k consumer)
(let ((reified-current-continuation (lambda (kl v) (k v))))
(consumer k reified-current-continuation))
```

cps-call/cc - функция с двумя параметрами: первый параметр - текущие остаточные вычисления, второй - функция, аналогичная функции в *call/cc.*

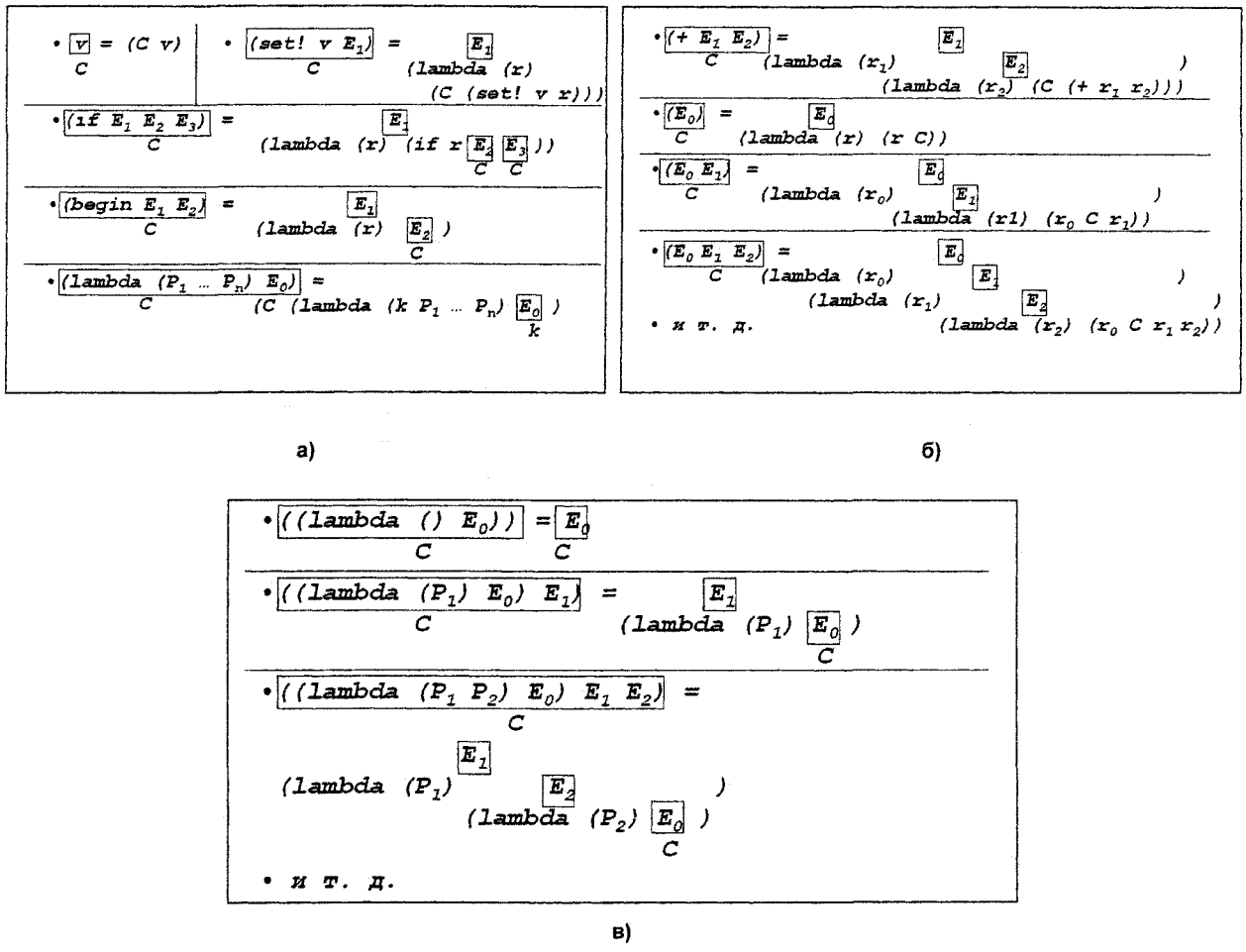

Рис. 13. Правила CPS-преобразований

Внутри *cps-call/cc* определяется функция *reified-current-continuation*, которая принимает на вход два аргумента: первый *k1* – текущие остаточные вычисления, он не используется, так как происходит возврат к сохраненным остаточным вычислениям, результат второго параметра *v* будет передан остаточным вычислениям, которые были сохранены в момент вызова *cps-call/cc (k v)*.

Рассмотрим пример.

(call/cc (lambda (cont) (cont 8)))

преобразуется в

```
(call/cc-cps
halt
(lambda (k cont) (cont k 8)))
```

Рассмотрим подстановку в определение *cps-call/cc*. Вместо *k* подставляется *halt*, вместо *consumer – (lambda (k cont) (cont k 8))*. Эволюция будет происходить следующим образом:

```
((lambda (k cont) (cont k 8))          ;consumer
   halt                                ;k
   (lambda (k1 v) (halt v)));reified-current-continuation
((lambda (k1 v) (halt v))              ;cont
   halt 8)                             ;параметры k и 8
3) (halt 8)                            ; результат программы: 8
```

5.5. Результат преобразований, решение проблемы остаточных вычислений

Итак, большинство современных компиляторов языка Scheme, работают по следующему принципу: сначала к программе применятся CPS-преобразование, разрешающее проблему остаточных вычислений и позволяющее эффективно работать с ними, как с объектами первого класса. Затем к полученной программе применяется преобразование замыканий, решающее проблему

свободных переменных. Заметим, что эти преобразования совершаются средствами самого языка, без введения дополнительных промежуточных представлений, выходящих за рамки Scheme.

6. Трансляция в С

Для завершенности картины кратко рассмотрим генерацию кода. На данном этапе имеется программа, преобразованная в CPS, к которой применены правила преобразования в замыкания. То есть существует набор замыканий, переход между которыми может осуществляться простым прыжком на тело очередного замыкания. Такая форма программы называется *closure-passing style*. При вызове следующего замыкания, мы должны передать ему параметры и совершить переход на его тело. Для этого необходимо научиться расставлять метки и решить, как мы будем передавать параметры. Подробнее о том, как это можно сделать, написано в [9,10].

Рассмотрим трансляцию кода на примере Gambit. Это один из самых мощных и быстрых интерпретаторов. В качестве промежуточного механизма в Gambit используется виртуальная машина (GVM), каждая команда которой представляет собой **макрос языка С.** Для передачи параметров служат регистры, в которых могут храниться объекты любых необходимых типов. Перед вызовом операции в регистр R0 загружается метка, на которую необходимо перейти после выполнения операции. В регистры R1..RN загружаются параметры N-местной операции. Для передачи параметров функциям, определенным программистом, используется 3 регистра и стек. Глобальные переменные и метки заносятся в таблицы с помощью специальных команд. В начале каждой функции ставится глобальная метка, с помощью которой можно вызвать функцию. После завершения работы операции результат возвращается в регистре R1.

Виртуальная машина предоставляет удобный интерфейс для работы с объектами в памяти. На более глубоком уровне работа с памятью также не предоставляет сложности. Все объекты располагаются в куче, и ее часть отдана под стек. В связи с тем, что должна происходить сборка мусора, объекты делятся на три типа: существующие все время - не удаляемые сборщиком мусора, располагающиеся на одном месте и перемещаемые. В зависимости от типа они помещаются в определенные области кучи. Из указателей на них могут формироваться списки, необходимые для работы сборщика мусора. Подробнее работа сборщика мусора описана в [10, 11].

7. Пример

Приведем пример, демонстрирующий все преобразования от начала до конца. Исходный текст:

```
(define square (lambda (x) (* x x))
(+ (square 5) 1)
```

После CPS-преобразования:

```
(define square (lambda (r1 x) (r1 (* x x))))
(square (lambda (r3)
        (let ((r2 (+ r3 1))) (halt r2))) 5)
```

После преобразования замыканий:

```
(define square (vector (lambda (self1 r1 x)
                 ((vector-ref r1 0) r1 (* x x)))))
((vector-ref square 0) square (vector (lambda (self2 r3)
                 (let ((r2 (+ r3 1))) (halt r2)))) 5)
```

После преобразования в С:

#include "gambit.h" – содержит описание всех используемых макросов

… – некоторые определения, не существенные для данной статьи

Далее идут команды виртуальной машины, которые являются макросами языка С. Их расшифровка представляет собой чисто техническую задачу, и любознательный читатель может посмотреть исходный код Gambit Scheme.

BEGIN_P_COD – начало программы

DEF_GLBL (L_MAIN) – метка на начало программы

SET_STK (1, R0) – поместить в вершину стека регистр R0

SET_R1 (FIX (5L)) – положить в регистр R1 значение 5

SET_R0 (LBL (1)) – после выполнения операции совершим переход на метку 1

OP_JMP (PARAMS (2), G_SQUARE) – выполнить двухместную операцию G_SQUARE и перейти на метку, содержащуюся в R0

DEF_SLBL (1, _MAIN_1) – локальная метка в теле программы с номером 1

SET_R2 (FIX (1L)) – положить в регистр R2 значение 1

SET_R0 (STK (-1)) – взять из стека метку конца программы и положить ее в R0

OP_JMP (PARAMS (2), _plus) – сложить значения в регистрах R1 и R2, результат поместить в регистр R1 и прейти по метке в регистре R0 (конец программы)

END_P_COD – конец программы

BEGIN_P_COD – начало функции SQUARE

DEF_GLBL (L_SQUARE) – метка на начало функции SQUARE

SET_R2 (R1) – поместить в регистр R2 такое же значение как и в R1

OP_JMP (PARAMS (2), mul) – перемножить регистры R1 и R2, результат положить в регистр R1 и перейти по метке в регистре R0 (на метку _MAIN_1)

END_P_COD – конец функции SQUARE

8. Заключение

В статье полностью показан процесс, основанный на *closure-passing style* и переводящий любую программу, написанную на языке Scheme, в программу на языке C. Это не единственный способ, но нам он показался наиболее эффективным, так как по этому принципу работают одни из самых быстрых интерпретаторов (Gambit, Chicken). Альтернативный подход, основанный на *environment passing style*, описан в [8, 12].

Кроме рассмотренных здесь проблем, существуют еще два аспекта – это сборка мусора и оптимизация. Подробнее об этом можно прочитать в статьях [10, 11]. Но, несмотря на то, что мы не рассказали об этих аспектах, приведенного здесь материала достаточно для написания собственного компилятора или интерпретатора языка Scheme [4].

Литература

1. Hudak, P. Conception, evolution, and application of functional programming languages / P. Hudak // ACM Computing Surveys. –1989. – Vol. 21. – Issue 3. – P. 359–411.

2. Haskel [HTML]: [http://www.haskell.org/]

3. The Computer Language Shootout Benchmarks [HTML]: [http://shootout.alioth.debian.org/]

4. Feeley M. The 90 minute Scheme to C compiler. Universite de Montreal. [HTML]: [http://www.iro.umontreal.ca/~boucherd/mslug/meetings/20041020/minutes-en.html]

5. Гринев, М. XML-СУБД Sedna: технические особенности и варианты использования / М. Гринев, С. Кузнецов, А. Фомичев // Открытые системы. – 2004. – №. 8. [HTML]: [http://www.osp.ru/os/2004/08/036.htm]

6. SXML [HTML]: [http://okmij.org/ftp/Scheme/SXML.html]

7. Revised (5) Report on the Algorithmic Language Scheme [HTML]: [http://www.swiss.ai.mit.edu/ftpdir/scheme-reports/r5rs-html/r5rs_toc.html]

8. Friedman, D.P. Essential Of Programming Languages, second edition / D.P. Friedman, M. Wand, C.T. Haynes. – The MIT Press, 2001.

9. Ctinger, W.D. Implementation Strategies for Continuations / W.D. Ctinger, E.M. Ost. – 1988 CM 0-09791-273-X/88/0007/0124.

10. Krishnamurthi, S. Continuations, 2001–10–12.

11. Appel, A.W. A Runtime System. Princeton University, CS-TR-220-89, May 1989.

12. Queinnec, C. Lisp in Small Pieces. – Cambridge University Press, 1996.

Поступила в редакцию 7 марта 2007 г.