© 2009 г. И. В. Сорокин

СПбГУ ИТМО, кафедра «Компьютерные технологии»

Использование автоматов для вычислений в комбинаторах

Введение

Последние редакции стандартов современных языков программирования *C++* и *C#* содержат элементы функционального программирования. Стандарт *C++0x* [1, 2] вводит в язык лямбда-выражения и новый механизм автоматического вывода типов. Язык *C# 3.0* [3] также включает в себя механизм вывода типов для локальных переменных и лямбда-выражения. Существует также языки программирования (например, *Nemerle*, *F#* и *Boo*), сочетающие в себе как элементы функционального, так и императивного программирования.

При этом возникает задача быстрого исполнения программ, написанных на функциональных языках программирования. Для этого было предложены различные модели вычисления лямбда-выражений. Одной из них являются комбинаторы. Они позволяют избавиться в лямбда-выражениях от абстракции, и тем самым существенно упростить и ускорить процесс вычисления.

Лямбда-исчисление

Лямбда выражение [4–8] можно рекурсивно определить как:

- `x` – переменная, где `x` – произвольный идентификатор;
- (`λx. E`) – абстракция, где `x` – произвольный идентификатор и `E` – произвольное лямбда-выражение. Абстракция представляет собой анонимную функцию аргумента `x`;
- $E_1\ E_2$ – аппликация, где E_1, E_2 – произвольные лямбда-выражения. Аппликация представляет собой применение E_1 к E_2.

Функцию V_f, сопоставляющую лямбда-выражению множество переменных, определим следующим образом:

- $V_f(x) = \{x\}$, где x – переменная;
- $V_f(\lambda x.E) = \{V_f(E) \backslash \{x\}\}$, где E – некоторое лямбда-выражение;
- $V_f(E_1E_2) = \{V_f(E_1) \cup V_f(E_2)\}$, где E_1, E_2 – некоторые лямбда-выражения.

Результат функции $V_f(E)$ называется множеством *свободных переменных* E.

Подстановкой $E[x := E']$ в лямбда-выражениях называется:

- $x[x := E'] = E'$;
- $y[x := E'] = y$, если x и y различны;
- $(E_1 E_2)[x := E'] = (E_1[x := E'] E_2[x := E'])$;
- $(\lambda x.E)[x := E'] = (\lambda x.E)$;
- $(\lambda y.E)[x := E'] = (\lambda y.E[x := E'])$, если x и y различны и $y \notin V_f(E')$;
- $(\lambda y.E)[x := E'] = (\lambda y'.E[y := y'])[x := E']$, если x и y различны и $y' \notin V_f(E')$.

α-конверсией называется подстановка вида:

$$E[x := x'], \text{ где } x' \notin V_f(E).$$

β-редукцией называется применение функции:

$$((\lambda x.E_1)E_2) = E_1[x := E_2].$$

η-конверсией называется следующее преобразование:

$$(\lambda x.fx) = f.$$

Заметим, что функции слева и справа возвращают одно и то же значение для всех своих аргументов.

Для записи лямбда-выражений будем использовать следующую нотацию. Применение функции f к аргументу x будем обозначать как $f\ x$, опуская скобки вокруг аргумента. Такая запись общепринята.

Аппликацию будем считать лево-ассоциативной:

$$f\ x\ y = ((f\ x)\ y).$$

Под применением *β*-редукции будем понимать отыскание в выражении конструкции вида $((\lambda x.E_1)E_2)$ и ее преобразование в соответствии с правилом приведенным выше. Такие конструкции называются *редексами* (***Red****ucible* ***Ex****pression*). В выражении может быть несколько редексов. В этом случае для данного выражения возможно сделать несколько разных β-редукций. Впрочем, таких конструкций может и не содержаться в

выражении. Будем говорить, что выражение находится в *нормальной форме*, если в нем нельзя сделать ни одной β-редукции.

Пусть задано некоторое выражение E_0. В нем можно выполнить некоторую β-редукцию – получилось выражение E_1. В нем сделаем еще одну β-редукцию – получилось выражение E_2. Пусть на некотором шаге получилось выражение E_n. Если E_n находится в нормальной форме, то E_n называют нормальной формой выражения E_0. Процесс получения нормальной формы выражения называют *вычислением* выражения. Можно доказать, что для данного выражения существует единственная нормальная форма. Если никакая из цепочек β-редукций не заканчивается выражением в нормальной форме – какую бы последовательность β-редукций не произвести, всегда можно сделать еще одну β-редукцию. В этом случае говорят, что выражение не имеет нормальной формы.

Лямбда-выражение можно представить в виде дерева, где внутренними элементами дерева могут быть аппликации и абстракции, а листьями – переменные. Левым потомком абстракции будет ее аргумент, а правым – тело. Левым потомком аппликации является функция, а правым – ее параметр.

Рассмотрим *left-first* обход дерева [9], который, если текущий элемент дерева является редексом, останавливается, сообщая, что здесь можно сделать β-редукцию. Можно показать, что если на каждом шаге делать β-редукцию, сообщенную этим обходом, то в случае, если лямбда-выражение имеет нормальную форму, рано или поздно можно прийти к ней.

Комбинаторы

Комбинаторное выражение определим следующим образом:

- x – переменная, где x – произвольный идентификатор;
- P – некоторая функция;
- $E_1\ E_2$ – аппликация, где E_1, E_2 – произвольные лямбда-выражения. Аппликация представляет собой применение E_2 к E_1.

Примеры комбинаторов

Простейшим является тождественный комбинатор, определяемый как функция одного аргумента, возвращающая переданное ей значение. Это можно записать следующим образом:

$$I\ x = x.$$

Другим простейшим комбинатором является комбинатор K, который позволяет создавать константные функции. $(K\ x)$ является функцией, всегда возвращающей x:

$$K\ x\ y = x.$$

Комбинатор S называется комбинатором обобщенной аппликации:

$$S\ x\ y\ z = x\ z\ (y\ z).$$

Заметим, что I выражается через комбинаторы S и K.

Действительно,

$$(S\ K\ K)\ x = S\ K\ K\ x = K\ x\ (K\ x) = x.$$

Таким образом,

$$(S\ K\ K)\ x = I\ x.$$

Хотя сами выражения $S\ K\ K$ и I не эквивалентны, не существует способа различить эти две конструкции в выражении. Такие выражения будем называть *экстенсионально эквивалентными.* (extensionally equal)

Заметим, что комбинаторы S, K, I представимы в виде лямбда-выражений:

$$I = (\lambda x.x);$$

$$K = (\lambda x\ y.x);$$

$$S = \left(\lambda x\ y\ z.x\ z\ (y\ z)\right).$$

Любое лямбда-выражение представимо в виде выражения, содержащего лишь S, K и I комбинаторы. Рассмотрим следующее преобразование $T[\ ]$:

- $T[x] \rightarrow x$;

- $T[E_1 E_2] \to T[E_1]\, T[E_2]$;
- $T[\lambda x.E] \to K\, T[E], если\ x \notin V_f(E)$;
- $T[\lambda x.x] \to I$;
- $T[\lambda x.\lambda y.E] \to T\left[\lambda x.T[\lambda y.E]\right], если\ x \in V_f(E)$;
- $T[\lambda x.(E_1 E_2)] \to S\, T[\lambda x.E_1] T[\lambda x.E_2]$.

Этот процесс также называется процессом исключения абстракции (*abstraction elimination*).

Набор комбинаторов, необходимый для представления любой программы, называется *базисом*. Комбинаторы S и K образуют базис. Также базисами являются (C, B, W, K), (C, B, S, I), где

$$B\ x\ y\ z = x\ (y\ z);$$

$$C\ x\ y\ z = x\ z\ y;$$

$$W\ x\ y = x\ y\ y.$$

Также интерес представляет *Y*-комбинатор:

$$Y\ f = f\ (Y\ f).$$

Y-комбинатор можно также записать как:

$$Y = \lambda f\ \left(\lambda x.f(x\ x)\right)\left(\lambda x.f(x\ x)\right).$$

Такая форма Y-комбинатора была предложена Карри Хаскеллом (*Haskell B. Curry*). Y-комбинатор может быть представлен в S, K, I виде:

$$Y = S\left(K\ (S\ I\ I)\right)\left(S\ (S\ (K\ S)K)\left(K\ (S\ I\ I)\right)\right).$$

Комбинатор $(S\ I\ I)$ также называется ω-комбинатором:

$$\omega\ x = (S\ I\ I)x = I\ x\ (I\ x) = \ x\ (I\ x) = \ x\ (x) = \ x\ x.$$

На языке лямбда-выражений этот комбинатор имеет вид:

$$\omega = (\lambda x.\ x\ x).$$

Вычисление выражений из комбинаторов

Рассмотрим выражение, записанное в *SKI*-базисе. В этом выражении все комбинаторы можно заменить на соответствующие лямбда-функции. Полученное выражение можно вычислить. Однако выражения из комбинаторов можно вычислять быстрее.

Заметим, что исходное выражение из комбинаторов не содержит абстракций. Для того, чтобы сделать β-редукцию, необходимо иметь аппликацию, первым аргументом которой является абстракция. Поэтому, по своей сути, комбинатор, к которому что-то применяют, является местом, где можно сделать β-редукцию. Рассмотрим пример *I*-комбинатора:

$$I\ a = (\lambda x.\ x)a = a.$$

От этого комбинатора можно не переходить к лямбда-выражениям, а сделать редукцию сразу. Аналогично для S- и *K*-комбинаторов. Для того, чтобы полученное в результате выражение оставалось комбинаторным, необходимо, чтобы аргументов для этих комбинаторов было достаточно.

Также как и для лямбда-выражений, будем говорить, что выражение находится в нормальной форме, если в нем нельзя применить ни одного комбинатора. Это происходит, когда ни одному комбинатору в выражении недостаточно аргументов для того, чтобы быть примененным.

Поэтому можно предложить следующий способ вычисления комбинаторных выражений. Выбирается любой комбинатор в выражении, после этого необходимо убедиться, что ему передано достаточное число аргументов и делается преобразование в соответствии с определением комбинатора.

При этом, так же, как и с лямбда-выражениями, если на каждом шаге брать самый левый комбинатор, то, если нормальная форма исходного выражения существует, то рано или поздно её можно получить.

Рассмотрим теперь вычисление выражения $S\ f\ g\ x$:

$$S\ f\ g\ x = f\ x\ (g\ x).$$

Здесь f и g некоторые функции. Если x содержит внутри себя некоторые нетривиальные действия и его вычисление может занимать время, то теперь получается, что x будет необходимо вычислять дважды. Первый раз в составе $(f\ x)$, а второй – в $(g\ x)$.

Можно ли это сделать один раз? Можно. Для этого необходимо, чтобы S-комбинатор не копировал значения x, а в те места, где должен был быть размещен x, он ставил ссылку на x. Таким образом, в том месте, где первый раз потребуется обращение к x, он будет упрощен, и если он потребуется еще раз, то будет использовано уже упрощенное значение.

Построим, используя эту идею, простейший вычислитель комбинаторных выражений. При этом необходимо реализовать концепцию вычисления без перехода к лямбда-выражениям и без вычисления одних и тех же выражений дважды. В процессе работы также требуется простым способом определять достаточно ли аргументов для того комбинатора, который преобразуется в данный момент. Вычислитель будет работать как интерпретатор [10].

Рассмотрим некоторое выражение из комбинаторов $\alpha_1\alpha_2 \dots \alpha_n$, которое не содержит скобок. Так как аппликация лево-ассоциативна, то это выражение эквивалентно выражению $\left(\left((\alpha_1\alpha_2)\alpha_3\right)\dots\alpha_n\right)$. Если α_1 является I-комбинатором и $n \geq 2$, то это выражение эквивалентно выражению $\alpha_2\alpha_3 \dots \alpha_n$. Действительно,

$$\left(\left((\alpha_1\alpha_2)\alpha_3\right)\dots\alpha_n\right) = \left(\left((I\,\alpha_2)\alpha_3\right)\dots\alpha_n\right) = \left((\alpha_2\alpha_3)\dots\alpha_n\right).$$

Аналогично для K-комбинатора при $n \geq 3$: $\alpha_2\alpha_4\alpha_5 \dots \alpha_n$. Для S-комбинатора при $n \geq 4$: $\alpha_2\alpha_4(\alpha_3\alpha_4)\alpha_5 \dots \alpha_n$.

Заметим, что рассмотренные преобразования затрагивают только начало выражения. Следует обратить внимание также на то, что $\alpha_2\alpha_3 \dots \alpha_n$ могут быть произвольными. Это могут быть как комбинаторы, так и целые выражения. В случае, если n меньше числа аргументов, необходимого для проведения β-редукции комбинатора α_1, то данное выражение больше не упрощается.

Что делать, если α_1 не комбинатор, а выражение? Пусть $\alpha_1 = \beta_1\beta_2 \dots \beta_n$ выражение, которое не содержит скобок. Тогда

$$\left(\left((\alpha_1\alpha_2)\alpha_3\right)\dots\alpha_n\right) = \left(\left(\left(\left(\left((\beta_1\beta_2)\beta_3\right)\dots\beta_n\right)\,\alpha_2\right)\alpha_3\right)\dots\alpha_n\right).$$

Это означает, что исходное выражение эквивалентно выражению вида $\beta_1\beta_2 \dots \beta_n\alpha_2\alpha_4\alpha_5 \dots \alpha_n$.

Таким образом, можно вычислять выражения, которые не содержат скобок, элементами которых могут быть комбинаторы и другие выражения, не содержащие скобок. Однако, если выражение содержит скобки, то его можно привести к бесскобочному виду.

Пусть дано выражение, содержащее скобки $\alpha_1\alpha_2(\beta_1\beta_2\ldots\beta_m)\ldots\alpha_n$. Обозначим $(\beta_1\beta_2\ldots\beta_m)$ через γ. При этом исходное выражение преобразуется к виду: $\alpha_1\alpha_2\gamma\ldots\alpha_n$. В случае, если выражение γ будет содержать скобки внутри себя, то применим этот же подход к нему.

Заметим, что, так как на каждом шаге у вычисляемых выражений изменяется только начало, то для их хранения можно использовать структуру данных типа «стек». Дном стека будет конец строки, а вершиной – начало.

Рассмотрим следующую модель вычислителя комбинаторных выражений, предложенную автором. Вычислитель представляет собой `N`-стеков. Элемент в вершине стека будем называть первым элементом данного стека. Элемент под ним – вторым и т.д. При этом `i`-ый элемент `j`-го стека будем обозначать α_i^j. Будем обозначать число элементов в данном стеке как C_j. Набор стеков вычислителя можно изобразить графически:

$$\begin{cases} \alpha_1^1\alpha_2^1\cdots\alpha_{C_1}^1 \\ \alpha_1^2\alpha_2^2\cdots\alpha_{C_2}^2 \\ \quad\vdots\qquad\quad\vdots \\ \alpha_1^N\alpha_2^N\cdots\alpha_{C_N}^N \end{cases}$$

Элементами этих стеков могут быть комбинаторы (S, K, I), ссылки на стеки, а также константы. Стек из элементов $\alpha_1^j\alpha_2^j\cdots\alpha_{C_j}^j$ будет соответствовать выражению $\left(\left(\left(\alpha_1^j\alpha_2^j\right)\alpha_3^j\right)\cdots\alpha_{C_j}^j\right)$.

Например, выражение $S\left(K\left(S\,I\,I\right)\right)I$ имеет следующее представление:

$$\begin{cases} 1: S\,L_2\,I \\ 2: K\,L_3 \\ 3: S\,I\,I \end{cases}$$

Исходному выражению соответствует первый стек. L_i означает ссылку на `i`-й стек.

Некоторый, например первый, стек должен соответствовать вычисляемому выражению. Для вычисления выражения вычислитель может взять первый элемент любого стека и сделать ему редукцию. Если первый элемент стека – это ссылка на какой-то другой стек, то вычислитель может подставить его содержимое на место ссылки. Такую операцию имеет смысл делать только в том случае, если в стеке, на который эта ссылка указывает, уже нельзя сделать β-редукцию. Заметим, что редукции можно проводить в любом порядке. Вычисление выражения останавливается, когда во всех стеках либо на первом месте стоит константа, либо недостаточно элементов в качестве аргументов для осуществления редукции.

Построим автомат, производящий такие редукции. Множеством событий такого автомата будут различные комбинаторы. Множеством выходных воздействий будут правила, по которым преобразуются верхние элементы стеков.

Схематично автомат для вычисления выражений представлен на рис. 1.

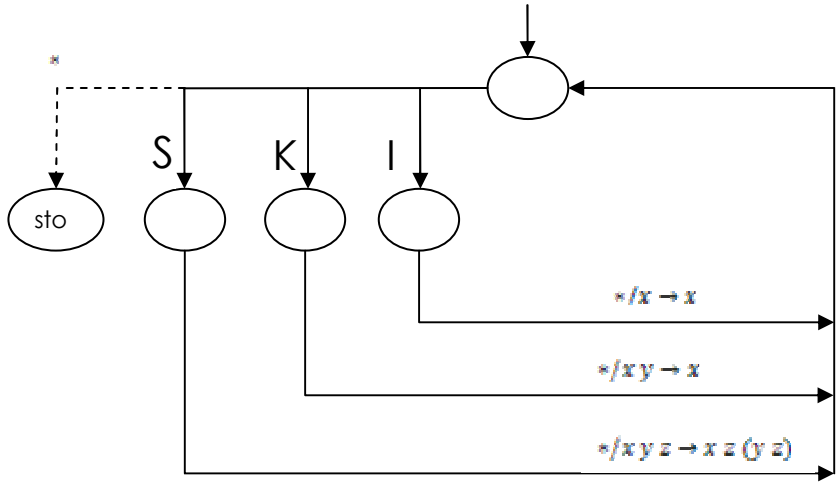

Рис. 1. Автомат для вычисления выражений, содержащих комбинаторы *S, K, I*

Заметим, что при создании интерпретатора не имеет смысла ограничивать себя минимальным набором комбинаторов. Существуют функции, вычисление которых в *SKI*-базисе занимает большое число шагов, но их можно вычислять быстрее, если не ограничивать себя фиксированным набором комбинаторов. Например, для вычисления функции `(S(KS)K)`, требуется пять шагов:

$$(S(KS)K)\ x\ y\ z = S\ (KS)K\ x\ y\ z = K\ S\ x\ (K\ x)y\ z = S\ (K\ x)y\ z = K\ x\ z\ (y\ z) = x\ (y\ z)$$

Однако, имея более широкий набор базисных комбинаторов, ее можно было бы вычислить за меньшее число шагов.

Рассмотрим теперь вычисление выражения $S\ K\ x\ y$:

$$S\ K\ x\ y = K\ y\ (x\ y) = y.$$

Оно требует от автомата выполнения двух действий. Можно доработать автомат для того, чтобы он вычислял данное выражение за один шаг. Этот автомат приведен на рис. 2.

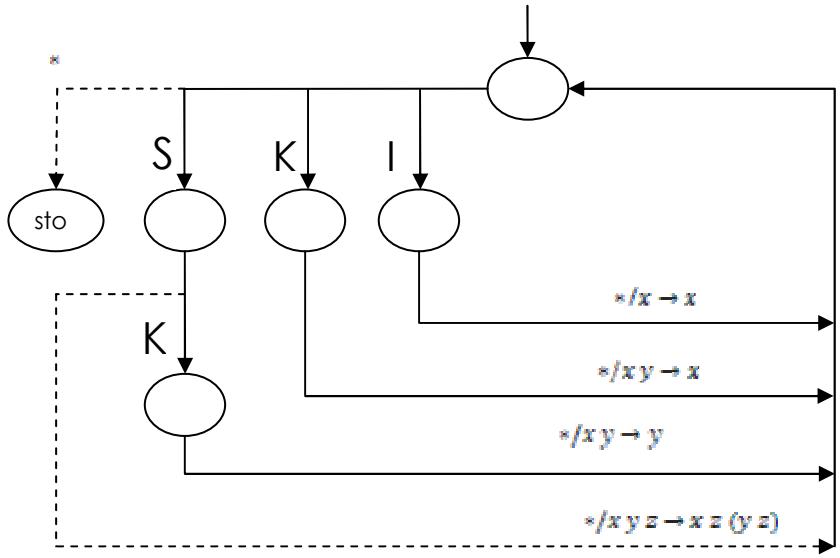

Рис. 2. Автомат для вычисления выражений, содержащих комбинаторы *S, K, I,* который вычисляет последовательность *S K* за один шаг

Такой автомат умеет легко редуцировать цепочки I, K, S и $S\ K$. Его можно легко расширять, добавляя новые правила редукций.

Отметим, что в автомат была добавлена новая цепочка $S\ K$.

Можно ли сделать так, чтобы автомат, работая, умел модифицировать себя, добавляя к самому себе новые цепочки? Действительно, встретив один раз цепочку $S\ K$, почему бы не модифицировать автомат таким образом, чтобы в следующий раз вычислить ее за один шаг.

То, сколько аргументов для редукции цепочки γ требуется, вычисляется достаточно просто. Также просто определяется то, какое правило преобразования соответствует этой цепочке. Для этого достаточно проинтерпретировать выражение $\gamma\ \xi_1\xi_2\ \ldots\xi_n\ \ldots$, где $\xi_1\xi_2\ \ldots\xi_n\ \ldots$ – некоторые константы. При интерпретации необходимо запомнить максимум числа

констант из стека, которые были использованы. При этом то, в каком порядке они располагаются, будет результатом преобразования. Например,

$$S\,K\,\xi_1\xi_2\xi_3\ldots\xi_n\ \ldots = K\,\xi_2(\xi_1\xi_2)\xi_3\ldots\xi_n\ \ldots = \xi_2\xi_3\ \ldots\ \xi_n\ \ldots$$

В этом преобразовании использованы лишь две константы. Для вычисления выражения $S\,K$ необходимо иметь два аргумента и преобразование $\xi_1\xi_2 \to \xi_2$, совершаемое цепочкой $S\,K$. Это полностью совпадает с тем, как было определено поведение цепочки $S\,K$ для второго автомата.

Предположим, что такой адаптирующийся автомат запущен на некотором сложном выражении, и ему дали поработать достаточное время. Рассмотрим, какой вид будет иметь этот автомат. Этот автомат позволяет вычислять достаточно длинные цепочки комбинаторов за один проход. Он хорошо работает на выражениях вида $\left(\left((\alpha_1\alpha_2)\alpha_3\right)\ldots\alpha_n\right)$, если α_i – некоторые комбинаторы. Однако, если одно из α_i является выражением, то автомат редуцирует первые $i-1$ комбинаторов, а затем подставляет выражение соответствующее α_i.

После этого автомат продолжает вычисления дальше. Можно ли сделать так, чтобы эту операцию делать за один шаг? Можно. Пусть вычисляется выражение $\alpha_1\alpha_2(\beta_1\beta_2\ldots\beta_m)\ldots\alpha_n$. Если обозначить выражение $(\beta_1\beta_2\ldots\beta_m)$, как γ, и если бы автомат умел вычислять выражения, содержащие γ, то выражение можно вычислять за один шаг.

Поэтому всякий раз, когда к автомату добавляется новая цепочка $\beta_1\beta_2\ldots\beta_m$ комбинаторов, будем обозначать ее новой буквой γ. При этом во всех правилах тех цепочек, которые уже встречали, заменим $\beta_1\beta_2\ldots\beta_m$ на γ.

Таким образом, расширяется не только множества состояний и переходов автомата, но и множество его входных воздействий.

Для реализации такого подхода требуется наличие соответствующих структур данных, позволяющих быстро находить последовательности комбинаторов для их применения.

Программная реализация

Используя изложенное, была написана программа. Она принимает на вход лямбда-выражение и разбирает его, применяя *LL(1)*-парсер. Затем она

использует процесс исключения абстракции для преобразования выражения к комбинаторному виду и начинает его вычислять.

Комбинаторное выражение у этой программы может содержать три вида элементов: комбинаторы, аргументы и ссылки на другие выражения.

Комбинатор задается набором правил редукции. Каждое правило состоит из числа `n`, указывающего сколько аргументов необходимо для проведения редукции, и шаблона редукции. Этот шаблон представляет обычное комбинаторное выражение, в котором аргументы обозначают места, на которые необходимо подставить фактические аргументы, переданные комбинатору.

Например, у комбинатора *S* набор правил редукции состоит из единственного правила, у которого $n = 3$, а шаблон редукции имеет вид $\alpha_1 \alpha_3 (\alpha_2 \alpha_3)$.

Несколько правил редукции для комбинаторов требуются для представления комбинаторов, сгенерированных самой программой. Например, комбинатор $A = K\ S$, будет иметь два таких правила:

$$A\ \alpha_1 = S$$

$$A\ \alpha_1\ \alpha_2\ \alpha_3\ \alpha_4 = \alpha_2\ \alpha_4\ (\alpha_3\ \alpha_4)$$

При его применении будет выбираться правило с наибольшим числом аргументов, для которого достаточно аргументов.

Для вычисления выражения программа использует три операции: операция проведения одной редукции, операция упрощения выражения и операция полного упрощения выражения.

Операция проведения одной редукции получает на вход выражение, пытается провести в нем преобразование и сообщает вызывающей функции, удалось ли сделать упрощение. Для выбора того, какое преобразование применить, используется автомат. Программа проходит текущее выражение и, пока в нем встречаются комбинаторы, применяет их как события для автомата. Если выражение закончилось, встретился аргумент или ссылка, то применяется преобразование, соответствующее текущему состоянию автомата. Если в выражении встретился комбинатор, а из текущей вершины нет перехода, соответствующего данному комбинатору, то происходит процесс генерации нового комбинатора. Если выражение начинается со

ссылки, то вызывается операция упрощения выражения по ссылке, а затем выполняется подстановка результатов в исходное выражение.

Операция упрощения вызывает операцию проведения одной редукции до тех пор, пока та не сообщит, что выражение больше упростить нельзя.

Операция полного упрощения выражения вызывает операцию упрощения выражения, а затем для всех элементов-ссылок выражения вызывает себя же, и если получившийся результат состоит из одного элемента, подставляет его в исходное выражение.

Операция полного упрощения применяется для максимального упрощения выражения при выводе выражения пользователю. Ее нельзя использовать вместо операции упрощения, поскольку она может начать вычислять невычислимые выражения, которые для получения результата вычислять не требуется.

Операция генерации нового комбинатора выполняется следующим образом. Вначале в отдельный стек складываются все элементы, пройденные операцией проведения одной редукции. Им делается упрощение.

Если после проведения операции упрощения в начале стека лежит комбинатор, то это означает, что ему просто не хватило аргументов для того, чтобы быть вычисленным. Поэтому в стек добавляется число аргументов, необходимое для проведения редукции с самым меньшим числом аргументов у этого комбинатора. После этого делается операция упрощения. Полученный результат запоминается как правило редукции, для которого число аргументов, необходимых для нее, равно текущему числу добавленных аргументов, а шаблон редукции – текущее выражение в стеке. После этого опять делается проверка на наличие в начале стека комбинатора, и цикл повторяется.

Если после упрощения в начале стека находится не комбинатор, то операция генерации комбинатора останавливается, и полученный комбинатор добавляется к остальным комбинаторам. В автомат добавляется новое состояние и переходы к нему.

Тестирование

Для тестирования программы проводилось вычисление факториала на базе арифметики Черча [11].

Эта арифметика представляет собой аналог обычной арифметики, в которой числа представляются лямбда-функциями. Число `n` – это функция, которая получает два аргумента и применяет первый аргумент ко второму `n` раз:

$$zero = \lambda f\ x.x$$

$$one = \lambda f\ x.f\ x$$

$$two = \lambda f\ x.f\ (f\ x)$$

При этом операцию увеличения числа на единицу, можно представить следующим образом:

$$next = \lambda n\ f\ x.f\ (n\ f\ x)$$

Операцию сложения:

$$add = \lambda n\ m\ f\ x.n\ f\ (m\ f\ x)$$

Операцию умножения:

$$mul = \lambda n\ m\ f.n\ (m\ f)$$

Арифметика Черча допускает реализацию всех основных арифметических операций. Приведем программу, вычисляющую факториал, которую автор использовал для тестирования (вычисляется факториал девяти):

```
( \zero
   ( \one
      ( \next
         ( \add
            ( \mul
               ( \true
                  ( \false
                     ( \pair
                        ( \first
                           ( \second
                              ( \phi
                                 ( \prior
                                    ( \fix
                                       ( \iszero
                                          ( \fact
                                             fix fact (\f \x f (f (f (f (f (f
(f (f (f x))))))))) 1 2
                                          ) (\f \n (iszero n) (one) (mul n (f
(prior n))))
                                       ) (\n n (\x false) true)
                                    ) (\f (\x f (x x)) (\x f (x x)))
                                 ) (\n first (n phi (pair zero zero)))
                              ) (\x pair (second x) (next (second x)))
                           ) (\p p false)
                        ) (\p p true)
                     ) (\a \b \f f a b)
```

```
                    ) (\x \y y)
                ) (\x \y x)
            ) (\a \b \f a (b f))
        ) (\a \b \f \x a f (b f x))
    ) (\n \f \x f (n f x))
) (\f \x f x)
) (\f \x x)
```

Результаты

За счет генерации новых комбинаторов удалось получить выигрыш в скорости вычисления по сравнению случаем без использования дополнительных комбинаторов (таблица).

Таблица

Аргумент факториала	Длительность вычисления без использования дополнительных комбинаторов	Длительность вычисления с использованием дополнительных комбинаторов	Ускорение
8	0:08.5	0:07.8	9%
9	1:17.9	1:12.1	8%
10	12:40.9	11:46.7	8%

Использование автоматов [12–14], позволяет эффективно определять то, какой комбинатор необходимо применить в данном случае. При этом средний выигрыш по скорости составил 8%.

Тестирование сложно провести на более широком диапазоне аргументов, так как при меньших значениях вычисление происходит очень быстро и возможна ошибка, а при больших – вычисление занимает длительное время и требует большого объема оперативной памяти.

Как показало тестирование предложенного алгоритма, основной причиной, по которой выражения нельзя вычислять быстрее, является то, что, как правило, в каждом из стеков находится не очень большое число элементов. Обычно эта величина колеблется от двух до пяти. Сгенерированные комбинаторы обеспечили бы большую эффективность, если эта величина будет больше. Для этого можно воспользоваться техникой, которая была описана выше. При этом необходимо не просто заменять последовательность комбинаторов новым, но и во всех правилах редукции, где встречается эта последовательность, заменять ее на новый комбинатор.

Ссылки

1. http://en.wikipedia.org/wiki/C++0x
2. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2565.html
3. http://en.wikipedia.org/wiki/C_Sharp_(programming_language)
4. Barendregt H., Barendsen E. Introduction to Lambda Calculus http://www.cs.ru.nl/E.Barendsen/onderwijs/sl2/materiaal/lambda.pdf
5. Jung A. A Short Introduction to the Lambda Calculus http://www.cs.bham.ac.uk/~axj/pub/papers/lambda-calculus.pdf
6. Rojas R. A Tutorial Introduction to the Lambda Calculus http://www.mscs.dal.ca/~selinger/papers/proto.pdf
7. http://en.wikipedia.org/wiki/Lambda_calculus
8. http://ru.wikipedia.org/wiki/Лямбда-исчисление
9. http://pco.iis.nsk.su/ICP/Practice/dd8-3/node6.html
10. http://en.wikipedia.org/wiki/Interpreter_(computing)
11. http://en.wikipedia.org/wiki/Church_encoding
12. Шалыто А. А. Технология автоматного программирования. М.: МГУ. 2003. http://is.ifmo.ru/works/tech_aut_prog/
13. Шалыто А. А. Парадигма автоматного программирования http://is.ifmo.ru/works/_2007_09_27_shalyto.pdf
14. Шалыто А. А. Автоматно-ориентированное программирование http://is.ifmo.ru/works/_politeh.pdf