стров и распределения оперативной памяти ЭВМ, а также для выполнения некоторых других оптимизирующих действий при генерировании объектного кода, в частности, для уменьшения исполнения команд безусловного перехода в объектных программах.

СПИСОК ЛИТЕРАТУРЫ

1. Погребной В.К. Об одном способе представления алгоритмов в виде графовых моделей // Управляющие системы и машины. – 1983. – № 1. – С. 63–69.
2. Евстигнеев В.А. Применение теории графов в программировании / Под ред. А.П. Ершова. – М.: Наука, 1985. – 352 с.
3. Огородов С.В., Трегубова Е.В. О структурном анализе графовых моделей алгоритмов // Методы и программы решения оптимизационных задач на графах и сетях. Ч. 1: Алгоритмы, программы, применения: Тез. докл. II Всесоюз. совещ. – Новосибирск, ВЦ СО АН СССР, 1982. – С. 156-158.
4. Касьянов В.Н. Анализ структур программ // Кибернетика. – 1980. – № 1. – С. 48–61.
5. Майника Э. Алгоритмы оптимизации на сетях и графах. – М.: Мир, 1981. – 323 с.
6. Огородов С.В. Задача автоматизации распределения регистров на графовых моделях алгоритмов // Методы и программы решения оптимизационных задач на графах и сетях. Ч. 1: Алгоритмы, программы, применения: Тез. докл. III Всесоюз. совещ. – Новосибирск: ВЦ СО АН СССР, 1984. – С. 169–171.

Поступила 14.04.2008 г.

Ключевые слова:

Ориентированные графы, графовые модели программ, управляющий граф программы, ветви и ветви-связей.

УДК 681.142.2

ЯЗЫК ОПИСАНИЯ ГЕНЕРАТОРОВ КОМБИНАТОРНЫХ МНОЖЕСТВ

А.В. Титков, В.В. Кручинин

Томский государственный университет систем управления и радиоэлектроники
E-mail: avt@tcde.ru

Предложен универсальный инструмент для построения и исследования генераторов комбинаторных множеств, основанный на функциональном языке, позволяющем описывать схемы рекурсивных композиций деревьев И/ИЛИ. Предложен вариант реализации такого инструмента, как расширение системной библиотеки STL для языка программирования C++.

Введение

Алгоритмы генерации комбинаторных множеств находят широкое применение при построении и исследовании разнообразных программно-технических систем. Например, в различных системах тестирования. Термин «программный генератор» носит достаточно общее толкование, например «генератор отчета», «генератор программного кода», «генератор случайных чисел» [1]. Здесь под генератором будет пониматься программа, реализующая некоторый алгоритм генерации комбинаторного множества. Комбинаторное множество представляет собой конечное множество, элементы которого имеют некоторую общую структуру. Например, множество перестановок, разбиений, двоичных деревьев, таблиц специального вида и т. д.

Различают 4 класса алгоритмов комбинаторной генерации [2]:

1) последовательная генерация множества всех комбинаторных объектов данного класса (*listing*);
2) нумерация всех комбинаторных объектов данного класса (*ranking*);
3) генерация комбинаторных объектов в соответствии с процессом нумерации (*unranking*);
4) случайная генерация реализации комбинаторного объекта (*random selection*).

Наиболее изучены алгоритмы последовательной генерации. Однако всё более востребованными являются генераторы, в основе которых лежат алгоритмы нумерации объектов и генерации объекта по его номеру.

Методов построения алгоритмов генерации и нумерации существует большое множество, и все они зависят от конкретного рассматриваемого комбинаторного объекта. Сравнительно недавно появился универсальный метод построения алгоритмов генерации и нумерации комбинаторных объектов, основанный на представлении комбинаторного множества в виде дерева И/ИЛИ [3].

В данной статье предлагается на основе метода построения и исследования алгоритмов генерации комбинаторных множеств универсальный программный инструмент построения и исследования генераторов элементов комбинаторных множеств.

Основные этапы и объекты метода

Метод построения алгоритмов генерации и нумерации комбинаторных объектов, основанный на

представлении комбинаторного множества в виде схемы рекурсивной композиции построения дерева И/ИЛИ, заключается в следующем [3]:

1. Исследуется множество комбинаторных объектов для построения рекурсивной композиции дерева И/ИЛИ. Рекурсивную композицию можно описать следующей схемой [4]:

$$R(0) = D_1,$$
$$R(n+1) = S_{i_2,i_2,\ldots,i_k}(D, D_1, D_2, \ldots, D_k),$$
$$D_j = R(n), \quad j = \overline{1,k}.$$

где D_0 и D – фиксированные деревья, $\{i_m\}_{m=1}^{k}$ – множество листьев дерева D, которые будут заменены на дерево $R(n)$ – полученное на n-ом шаге.

Операция рекурсивной композиции показана на рис. 1.

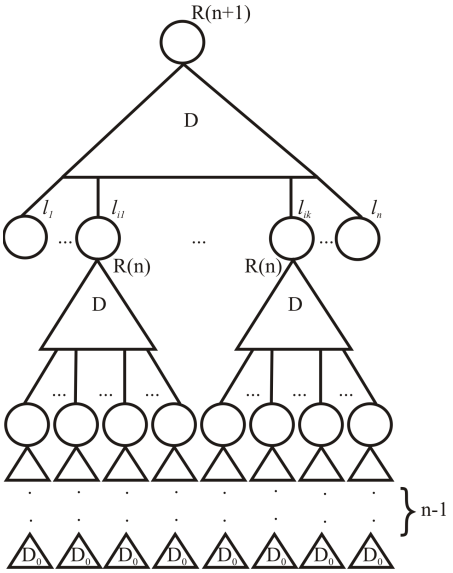

Рис. 1. *Операция рекурсивной композиции*

Показано, что если мощность комбинаторного множества задана в виде функции $f(n)$ алгебры $\{N,+,\times,R,S\}$, где R – оператор примитивной рекурсии, S – оператор суперпозиции, то для такого множества можно построить схему рекурсивной композиции деревьев И/ИЛИ.

2. Если рекурсивная композиция получена, то для построения алгоритма генерации объектов комбинаторного множества используется алгоритм генерации варианта дерева И/ИЛИ и далее по вариантам полученного дерева строятся объекты исследуемого множества [5]. Вариантом дерева И/ИЛИ является поддерево, которое получается из заданного путем отсечения выходных дуг кроме одной у всех ИЛИ-узлов.
3. Для алгоритмов нумерации по объекту строится вариант дерева И/ИЛИ и далее используя алгоритм нумерации варианта дерева получаем номер объекта в исследуемом множестве.

Таким образом, процесс построения генераторов комбинаторных объектов можно разделить на 3 части:

1. Реализация обобщённых алгоритмов деревьев И/ИЛИ (нумерации и генерации вариантов).
2. Реализация алгоритма построения дерева И/ИЛИ.
3. Получение варианта дерева И/ИЛИ по объекту в алгоритмах нумерации и получение объекта по варианту в алгоритмах генерации.

Первая часть может быть реализована в виде библиотеки шаблонов классов на языке C++. Это позволит применять библиотеку для различных типов комбинаторных объектов. Для решения второй части предлагается использовать специализированный язык, ориентированный на работу с деревьями И/ИЛИ. Это позволит уменьшить объем кода, необходимого для реализации генераторов, и соответственно уменьшить время разработки. Если первые две части задачи можно автоматизировать, то третья зависит от конкретного вида объекта и полностью ложится на плечи программиста.

Язык описания генераторов

Предлагаемый язык является функциональным, т. к. сам метод построения алгоритмов генерации основан на рекурсивной композиции. Язык позволяет:

1. Описать дерево И/ИЛИ в скобочной нотации или в виде рекурсивной композиции.
2. Производить над деревьями операции композиции, добавления и удаления узлов, производить доступ к данным в узлах деревьев.
3. Получать вариант дерева И/ИЛИ по его номеру и получать номер варианта по самому варианту.

Язык является интерпретируемым, а сам интерпретатор выполнен в виде библиотеки, что позволяет встраивать его в различные программные системы. Интерпретатор предоставляет интерфейс доступа ко всем элементам языка.

Язык оперирует только деревьями и функциями. Результатом выполнения любой операции или функции является дерево.

Все деревья в языке являются деревьями И/ИЛИ. Каждый узел дерева может быть 3-х типов: И-узел, ИЛИ-узел, лист. Узлы дерева могут быть именованными (существует идентификатор для узла) и неименованными (идентификатора не существует).

В качестве способа описания дерева выбрана скобочная запись:

```
d1=(l1,{l2,l1},l3);
```

Скобки `( )` означают узел «И», а скобки `{ }` узел «ИЛИ». `l1, l2, l3` – имена узлов. Дерево `d1` представлено на рис. 2.

Узлом дерева может быть математическое выражение:

```
d=(0, sin(0)+cos(x));
```

В этом случае математическое выражение вычисляется и именем узла становиться результат. В данном примере дерево `d` состоит из неименованного корня и двух листов `0` и `1` (т. к. `sin(0)+cos(0)=1`).

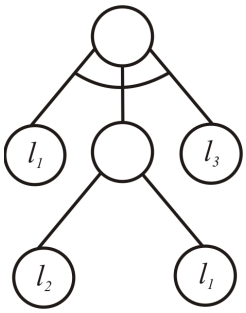

Рис. 2. *Дерево* `d1=(l1,{l2,l1},l3);`

В языке существует несколько операций над деревьями. Операции над деревом записываются в виде `имя_дерева[операции_над_деревом]`. Результатом любой операции является новое дерево. Исходное дерево не изменяется. Во всех операциях ветвь (узел или поддерево), над которой требуется произвести действие, может быть описана несколькими способами:

1) `leaf1` – описывает лист с именем `leaf1`.
2) `node1(...)` – описывает И-узел `node1` с произвольным количеством сыновей.
3) `node2{_, _, _}` – описывает ИЛИ-узел с именем `node2`, имеющий 3 сына. Знак подчёркивания означает присутствие узла с любым именем.
4) `node3(_x, _y)` – описывает И-узел `node3`, имеющий 2 сына. Здесь `_x` и `_y` – условные имена сыновей узла `node3` (отменой строгого соответствия имён узлов служит символ подчёркивания). Имена `_x` и `_y` могут быть использованы для манипулирования содержимым узлов.

Рассмотрим операции над деревьями.

Композиция деревьев.

Имеет следующий вид:

```
tree[branch -> tree1]
```

Все ветви `branch` дерева `tree` заменяются деревом `tree1`.

Различные виды композиции будут рассмотрены ниже.

Добавление сына к корню дерева.

Имеет следующий вид:

```
tree[+ tree1]
```

К корню дерева `tree` добавляется крайним правым сыном дерево `tree1`. Это возможно только в том случае, когда корень дерева `tree` не является листом. Дерево, стоящее в правом операнде, может быть описано непосредственно в самой операции:

```
tree[+ root1{node1, node2}]
```

Удаление узла дерева.

Имеет следующий вид:

```
tree[- branch]
```

Из дерева `tree` удаляется ветвь `branch`.

Функции доступа к узлам дерева.

`first(node)` – возвращает дерево, корнем которого является левый сын узла `node`.

`last(node)` – возвращает дерево, корнем которого является крайний правый сын узла `node`.

`parent(node)` – возвращает дерево, корнем которого является отец узла `node`.

`next(node)` – возвращает дерево, корнем которого сосед справа узла `node`.

Функции доступа к вариантам деревьев И/ИЛИ.

`vars(tree)` – возвращает количество вариантов дерева (ветви) `tree`.

`var(tree, num)` – возвращает дерево, которое является вариантом дерева (ветви) `tree` под номером `num`.

`vfirst(tree)` – возвращает первый вариант дерева (ветви) `tree`.

`vlast(tree)` – возвращает последний вариант дерева (ветви) `tree`.

`vnext(tree, var_tree)` – возвращает следующий за `var_tree` вариант дерева (ветви) `tree`.

Над деревом можно производить несколько операций подряд. Тогда эти операции записываются через запятую и выполняются в той последовательности, в которой записаны:

```
tree[leaf1->tree1,+tree1,leaf2->tree2]
tree[node1(...)->node1[+tree1],leaf1->tree2]
tree[node1{_, _}->tree1,node1(_x)->_x-1]
```

Язык позволяет описывать функции. Функция записывается в виде `Имя_Функции([Список_Аргументов]) = тело_функции`.

В качестве результата любая функция возвращает дерево. Функции могут быть двух видов: функция, возвращающая 1 узел (листовая функция), и функция, возвращающая дерево (древовидная функция).

Листовые функции.

Листовая функция возвращает один лист (вырожденное дерево), в котором записан результат выполнения функции. Тип узла далее может быть изменён при вызове функции в описании дерева.

Например:

```
F(x)=sin(x);
d=(F(0));
```

Здесь `F(x)` – функция, вычисляющая `sin(x)`; `d` – дерево, состоящее из неименованного корня и одного листа с именем `0` (т. к. `F(0)=sin(0)=0`).

Рассмотрим ещё один пример:

```
pi=3,14159265358979323846264338327795;
F(x)=sin(x)+1;
d=(F(0){cos(pi),sin(pi)+cos(pi)*(2+1)
(pi,2*pi)});
```

В данном примере узел `F(0)=0` является ИЛИ-узлом, а узел `sin(pi)+cos(pi)*(2+1)=-3` является И-узлом.

В итоге `d=(0 {-1, -3(3.14, 6.28)})`.

Древовидные функции.

Древовидные функции описывают некоторое дерево. Например:

```
pi=3,14159265358979323846264338327795;
F(x)={sin(x), cos(x), tg(x), ctg(x)};
```

При вызове `F(pi)` будет построено дерево, корнем которого будет неименованный узел, а в сыновьях будут находиться результаты вычисления функций `sin, cos, tg, ctg` от `pi`.

Рассмотрим ещё один пример:

```
C(m, n) = {C(m, n-1), C(m-1, n-1)};
C(m, m) = ;
C(0, n) = ;
```

В данном примере строиться дерево перестановок. Узлами дерева являются вызовы функции `C(m, n)`. Дерево будет строиться до тех пор, пока `m` не равно `0`, или `m` не равно `n`. Здесь запись `C(m,m)` означает, что оба параметра функции имеют одинаковое значение.

Ещё одной особенностью этого примера является то, что обычно узел, в котором происходит вызов какой-либо функции, принимает значение равное результату выполнения этой функции. Т. е. вместо узла-вызова подставляется узел-результат. Однако в том случае, когда функция является и *древовидной* и *рекурсивной*, то узел-вызов остается, а его сыном становится узел-результат. На рис. 3 представлен результат выполнения вызова `C(2, 4)`.

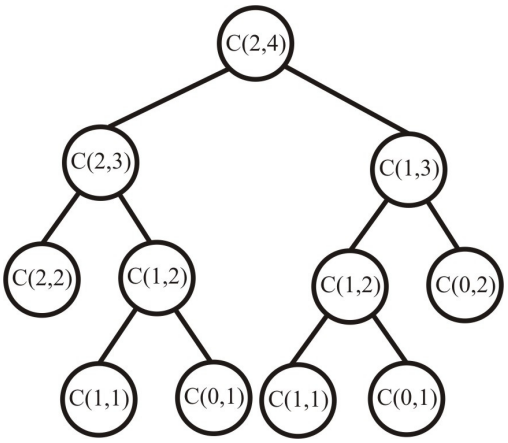

Рис. 3. *Результат выполнения вызова* `C(2, 4)`

Теперь рассмотрим виды композиций деревьев.

Простая композиция.

```
tree[branch -> tree1]
```

Все ветви `branch` дерева `tree` заменяются деревом `tree1`.

Пример:

```
d1 = (l1, {l2, l1}, l3);
d2 = {l4, (l1, l5)};
d1 [l1 -> d2];
```

Результат:

```
({l4,(l1,l5)},{l2,{l4,(l1,l5)}},l3)
```

Таким образом, все листы `l1` дерева `d1` заменены на дерево `d2`.

Рекурсивная композиция.

В языке нет оператора рекурсивной композиции. Однако эта операция легко реализуется при помощи рекурсивной функции:

```
d1 = (l1, {l2, l1}, l3);
d2 = {l4, (l1, l5)};
Composition(tree, l, t, 0) = tree;
Composition(tree, l, t, n) =
Composition(tree[l->t], l, t, n-1);
Composition(d1, l1, d2, 2);
```

Результат:

```
      ({l4,({l4,(l1,l5)},l5)},
  {l2,{l4, ({l4,(l1,l5)},l5)}},l3)
```

Суперкомпозиция.

Суперкомпозиция записывается как `tree [leaf1 -> tree1, leaf2 -> tree2]`. Это означает, что в дереве `tree` все листья `leaf1` будут заменены на дерево `tree2`, а листья `leaf2` на дерево `tree2`.

```
d1 = (l1, {l2, l1}, l3);
d2 = {l4, (l1, l5)};
d3 = (l6, l2);
d1 [l1 -> d2, l2 -> d3];
```

Результат:

```
({l4,(l1,l5)},{(l6,l2),{l4,(l1,l5)}},l3)
```

Рекурсивная суперкомпозиция.

Рекурсивная суперкомпозиция, как и рекурсивная композиция, реализуется с помощью рекурсивной функции:

```
d1 = (l1, {l2, l1}, l3);
d2 = {l4, (l1, l5)};
d3 = (l6, l2);
Func(tree, 0) = tree;
Func(tree, n) = Func(tree[l1->d2, l2-
>d3], n-1);
Func(d1, 2);
```

Результат:

```
      ({l4,({l4,(l1,l5)},l5)},
{(l6,(l6,l2)),{l4,({l4,(l1,l5)},l5)}},l3)
```

Предлагаемый язык можно широко использовать для изучения свойств различных множеств, так как сам метод построения алгоритмов генерации и нумерации комбинаторных объектов, положенный в основу языка, прекрасно для этого подходит.

Для генерации двоичных деревьев высотой не более n можно записать:

```
w(0) = ;
w(n)={w(0),w(n-1),w(n-1),(w(n-1), w(n-1))};
```

Здесь `w` – это рекурсивная функция, дерево вызовов которой и есть требуемое дерево.

Для генерации возрастающих деревьев можно записать:

```
   sum(low, high, cur, iter_func, tree) =
sum(low, high, cur+1, iter_func, tree[ +
iter_func(high, cur)]);
   sum(low, high, high, iter_func, tree)
= tree[+ iter_func(high, high)];
   sum(low, high, iter_func) = sum(low,
high, low, iter_func, {});
   iter_w(n, i)=(w(i), w(n-i), C(i, n));
   w(0) = ;
   w(n) = sum(0, n-1, iter_w);
```

С помощью предложенного языка можно записать все алгоритмы генерации, рассмотренные в [3].

Интерпретатор языка основан на библиотеке, реализующей шаблонный класс aotree. Данный класс является реализацией структуры данных «дерево И/ИЛИ» и является STL-совместимым. В нём реализованы алгоритмы подсчёта вариантов дерева И/ИЛИ, алгоритмы генерации и нумерации вариантов дерева И/ИЛИ [3], реализованы итераторы для нескольких видов обхода деревьев и итераторы для доступа к вариантам деревьев И/ИЛИ. Интерпретатор языка описания генераторов предоставляет доступ ко всем элементам программ, написанных на языке, что позволяет использовать интерпретатор в различных программных системах.

Рассмотрим пример использования интерпретатора языка описания генераторов в программе на языке C++ для создания генератора сочетаний. Генерируемые значения будут подаваться на вход тестовой системы:

```
Interpreter interpret;
string program = «\
C(m, n) = {C(m, n-1), C(m-1, n-1)};\
C(m, m) = ;\
C(0, n) = ;\
var(C(2, 4), 3);»;
aotree<Interpreter::TreeFunc>variant =
interpret.run(program);
TestSystem<<VariantConverter(variant);
```

В данном примере с помощью программы, написанной на языке описания генераторов, описывается рекурсивная композиция построения дерева И/ИЛИ для сочетаний, строится дерево И/ИЛИ `C(2, 4)` и из полученного дерева извлекается вариант дерева под номером 3. Полученный вариант помещается в переменную `variant` и, после преобразования классом `VariantConverter`, подаётся на вход тестируемой системы `TestSystem`.

Рассмотрим ещё один пример:

```
typedef   aotree<Interpreter::TreeFunc>
restree;
Interpreter interpret;
string program = «\
C(m, n) = {C(m, n-1), C(m-1, n-1)};\
C(m, m) = ;\
C(0, n) = ;\
C(2, 4);»;
restree tree = interpret.run(program);
for(restree::variator       v=tree.vbe-
gin();v!=tree.vend();v++)
TestSystem << VariantConverter(v);
```

В данном примере все варианты дерева сочетаний `C(2, 4)` получаются при помощи итератора вариантов класса aotree.

Как видно из привёденных выше примеров, использовать интерпретатор языка достаточно просто, объем кода становиться существенно меньше и понятней чем в случае прямого программирования данных примеров на языке C++.

Заключение

1. Предложен универсальный инструмент для построения и исследования генераторов комбинаторных множеств, который базируется на представлении комбинаторного множества в виде схемы рекурсивной композиции деревьев И/ИЛИ и алгоритмах генерации и нумерации вариантов.
2. Описан новый функциональный язык, позволяющий описывать схемы рекурсивных композиций деревьев И/ИЛИ и на основе их анализа, строить алгоритмы генерации и нумерации вариантов, соответствующих исследуемому комбинаторному множеству.
3. Предложен вариант реализации такого инструмента, как расширение системной библиотеки STL для языка программирования C++.

СПИСОК ЛИТЕРАТУРЫ

1. Кручинин В.В. Программные генераторы (обзор) // Доклады Томского государственного университета систем управления и радиоэлектроники. – 2004. – № 2 (10). – С. 64–68.
2. Ruskey F. Combination Generation. 2003. [www.1stworks.com/ref/RuskeyCombGen.pdf]
3. Кручинин В.В. Методы построения алгоритмов генерации и нумерации комбинаторных объектов на основе деревьев И/ИЛИ. – Томск: В-Спектр, 2007. – 200 с.
4. Кручинин В.В. Рекурсивные композиции и их свойства // Доклады Томского государственного университета систем управления и радиоэлектроники. – 2007. – № 2 (16). – С. 70–76.
5. Кручинин В.В., Титков А.В., Хомич С.Л. Подход к созданию баз данных, основанный на алгоритмах генерации и идентификации кортежей // Известия Томского политехнического университета. – 2006. – Т. 309. – № 8. – С. 28–32.

Поступила 28.03.2008 г.

Ключевые слова:

Комбинаторные множества, интерпретатор язывa, рекурсивная композиция.