Опубликовано в научно-техническом вестнике СПбГУ ИТМО. 2006, №25 .– с. 102–110.

ПОСТРОЕНИЕ МОДЕЛИ ДАННЫХ ПРОГРАММЫ ПО ИСХОДНОМУ КОДУ

Г.А. Корнеев
Научный руководитель А.А. Шалыто

В работе предлагается метод построения модели данных программы по ее коду на языке высокого уровня. В результате применения метода все переменные выносятся в модель данных, и строится модификация программы, использующая модель данных и эквивалентная исходной программе.

Введение

В рамках технологии автоматного программирования [1], рассматриваются формальные методы преобразования программ к автоматному виду [2, 3]. При этом для преобразования программы все используемые переменные должны быть вынесены в модель данных, предназначенную для сохранения вычислительного состояния.

В настоящей работе предлагается метод построения модели данных программы по ее исходному коду. При этом входными данными является исходный код программы на языке высокого уровня, а выходными данными — модель данных программы и модифицированный исходный код. В связи с наличием выходных данных двух типов, преобразование выполняется в два этапа:

1. *Построение модели данных* — по программе строится модель данных, в которую выносятся все используемые в программе переменные.
2. *Модификация программы* — программа модифицируется так, чтобы она использовала только переменные модели данных.

В первом разделе приведены требования к преобразуемой программе. Во втором — описывается построение модели данных для итеративных программ, а в третьем — для рекурсивных.

1. Требования к исходной программе

Сформулируем требования к исходной программе.

1. Параметры и возвращаемые значения процедур передаются по значению. Отметим, что это не исключает передачу ссылок или указателей.
2. В идентификаторах не должен использоваться символ подчеркивания (“_”).
3. Используемые выражения не должны иметь побочных эффектов — значения переменных изменяются только оператором присваивания.

Первое ограничение несущественно, так как все современные языки программирования позволяют возвращать из процедур объекты и/или структуры. В языках *Java*, *C* и *C++* все данные передаются по значению. В языке *Pascal* возможен обход этого правила, путем применения ключевого слова `var`. Однако, такие программы легко преобразуются к требуемому виду за счет применения указателей.

Второе ограничение наложено для того, чтобы символ подчеркивания мог быть использован для образования структурированных идентификаторов. При необходимости, переменные, используемые в программе, можно переименовать произвольным образом.

Третье ограничение исключает применение цепных операторов присваивания, а также операции инкремента (++) и декремента (––) в языках *Java*, *C* и *C++*. Цепные операторы присваивания легко преобразуются в последовательность присваиваний. В свою очередь, выражения, использующие операции инкремента и декремента, преобразуются в последовательность из нескольких выражений.

2. Построение модели данных по итеративной программе

В соответствии с синтаксисом большинства процедурных языков программирования все переменные, используемые в реализации алгоритма можно разделить на два класса:

- *глобальные* — определенные на уровне программы и доступные во всех процедурах;
- *локальные* — определенные в рамках процедуры и доступные только в ней.
- *аргументы процедур* — также определенные в рамках одной процедуры и доступные только в ней.

Отметим, что если переменным модели давать те же имена, что и исходным переменным, то возможно дублирование имен переменных модели, что не допустимо. Для решения этой проблемы предлагается использовать правила, изложенные ниже.

Правила выделения глобальных и локальных переменных различны и будут рассматриваться отдельно в разделах 2.1 и 2.2 соответственно.

2.1. Построение модели данных

В случае итеративных программ построение модели разбивается на четыре шага. На каждом из них в модель добавляется набор переменных.

На первом шаге в модель выносятся глобальные переменные. Так как имена всех глобальных переменных различны, то их можно вынести в модель с сохранением имен. Этот шаг может быть формализован следующим образом.

1. Для каждой глобальной переменной в модель добавляется переменная с именем, совпадающим с именем соответствующей глобальной переменной. Тип добавляемой переменной соответствует типу глобальной переменной.

Здесь и далее при формализации действий, выполняемых в рамках шага, им присваиваются номера для последующих ссылок.

На втором шаге выполняется перенос в модель локальных переменных. Работа с локальными переменными более сложна, так как в разных процедурах могут быть определены переменные с одинаковыми именами. Поэтому при построении модели используется следующий прием: если переменная `a` определена в процедуре `calcSum`, то соответствующая ей переменная модели будет иметь имя `calcSum_a`. При этом используется второе требование к исходной программе (раздел 1). Тип добавленной переменной соответствует типу локальной переменной. В общем случае этот шаг может быть формализован следующим образом.

2. Для каждой локальной переменной в модель добавляется переменная с именем вида:

   <*имя процедуры*>_<*имя переменной*>

Переходя к третьему шагу, отметим, что в случае итеративных программ для аргументов процедур можно использовать ту же схему именования, как и для локальных переменных, что может быть сформулировано следующим образом.

3. Для каждого формального аргумента процедуры в модель добавляется переменная с именем вида:

   <*имя процедуры*>_<*имя формального аргумента*>

   Тип переменной модели соответствует типу формального аргумента.

В заключение, на четвертом шаге для процедур, возвращающих значения, необходимо добавить в модель переменные для хранения возвращаемых значений.

4. Для каждой процедуры, возвращающей значения, в модель добавляется переменная с именем вида:

   <*имя процедуры*>_

   и типом, соответствующим типу возвращаемого значения.

Предложенная схема образования идентификаторов позволяет не только избежать дублирования имен переменных модели, что требуется синтаксисом языка, но и однозначно восстанавливать по имени переменной модели, информацию о том, по какой переменной и из какой процедуры она порождена.

Заметим, что на шагах два и три используется правило, не позволяющее в большинстве языков программирования иметь в одной процедуре локальные переменные и формальные параметры с одинаковыми именами. В языках, разрешающих такие именования, можно имена переменных модели, соответствующих формальным аргументам процедуры, начинать с символа подчеркивания. Таким образом, шаг три примет вид

3. Для каждого формального аргумента процедуры в модель добавляется переменная с именем вида:

   <имя процедуры>_<имя формального аргумента>

   Тип переменной модели соответствует типу формального аргумента.

2.2. Модификация программы

Модифицируем программу, так чтобы она использовала только переменные модели данных, построенной, как указанно в предыдущем разделе.

В дальнейшем, предполагается, что экземпляр модели доступен всем процедурам, как переменная с именем `data`.

Так как программа итеративная, то для каждой локальной переменной, формального параметра процедуры и возвращаемого значения можно выделить статический участок памяти. Фактически это и производится при построении модели данных (шаги 1–4 предыдущего раздела).

Для модификации программы совершим следующие действия.

1. Удалим объявления глобальных переменных.
2. Добавим объявление глобальной переменной `data`.
3. Так как имена глобальных переменных при вынесении в модель не изменились, то для использования глобальных переменных требуется добавить к ним префикс “`data.`” .
4. Если некоторые локальные переменные инициализируются одновременно с их объявлением, то разобьем каждое объявление на две части: объявление и инициализация.
5. Удалим объявления локальных переменных.
6. Ко всем обращениям к локальным переменным добавим префикс вида

   `data.`*<имя процедуры>_*

В результате выполнения этих шагов все обращения к локальным переменным будут заменены обращениями к переменным модели.

Последующие шаги преобразуют вызовы процедур.

7. Операторы, вызывающие одну и туже процедуру более одного раза, разбиваются так, чтобы ни одна процедура не вызывалась дважды в одном операторе.
8. Вызов процедуры в выражении разбивается на две части:
   - оператор вызова процедуры (с сохранением возвращаемого значения в соответствующей переменной модели);
   - исходное выражение, с заменой вызова процедуры на обращение к переменной вида `data.`*<имя процедуры>_*.
9. Вызовы процедур преобразуются следующим образом. Пусть вызывается процедура с *N* формальными аргументами. Обозначим имена этих аргументов следующим образом:

   аргумент1, *аргумент2*, ... *аргументN*.

   Реальные аргументы обозначим как

выражение1, *выражение2*, ..., *выражениеN*

соответственно. Тогда оператор вызова процедуры заменяется на N операторов, вида:

`data.`*<имя процедуры>_<аргументM>* = *<выражениеM>*`;`

где M пробегает значения от 1 до N и вызов процедуры без аргументов.

10. Обращения к формальным аргументам процедуры заменим обращением к соответствующим переменным модели.
11. Заменим каждый оператор возврата значения

    `return` *выражение*`;`

    на присваивание значения выражения переменной

    `data.`*<имя процедуры>*`_`

    и простой оператор возврата (без возвращаемого значения).
12. Заголовки всех процедур изменяются так, чтобы процедуры не имели параметров и возвращаемых значений.

Отметим, что программа является работоспособной, не только после выполнения всех шагов, но и после выполнения первых двух, трех, четырех, шести или семи шагов, что может быть использовано для проверки правильности производимых преобразований.

2.3. Упрощенная запись (@-нотация)

Для ссылок на переменные, вынесенные в модель данных удобно применять *@-нотацию*: запись вида `@a` будет означать обращение к переменный `a` модели данных.

Если переменная `a` является глобальной, то запись `@a` эквивалентна записи

```
data.a
```

Если же `a` — локальная переменная, то `@a` будет означать ссылку на соответствующую локальную переменную, вынесенную в модель данных. На пример, если в процедуре `calcSum` была определена переменная `a`, то в ней запись `@a` будет эквивалентна записи

```
data.calcSum_a
```

Применение @-нотации позволяет приблизить запись программ, с вынесенной моделью данных к исходной записи.

В дальнейшем, во всех примерах для обращения к переменным модели применяется @-нотация.

2.4. Пример построения модели данных и модификации программы

Рассмотрим изложенный метод на примере программы, вычисляющей максимальный из локальных минимумов в массиве натуральных целых чисел (для примеров здесь и в дальнейшем используется язык *Java*).

Исходная программа имеет следующий вид:

```
int[] a; // Массив, в котором осуществляется поиск

/** Подсчитывает указанный максимум */
void calc() {
  int m = 0;
  for (int i = 1; i < a.length - 1; i++) {
    if (isMin(a[i-1], a[i], a[i+1]) && a[i] > m) m = a[i];
  }
}

/** Возвращает, является ли b минимумом аргументов */
boolean isMin(int a, int b, int c) {
  int m = a > b ? a : b;
  return b == (m > c ? m : c);
}
```

Построим модель данных, как указано в разделе 2.1:

```
/** Модель данных */
Class Model {
  int[] a; // Глобальная переменная
```

```
  int calc_m, calc_i, isMin_m; // Локальные переменные
  int isMin_a, isMin_b, isMin_c; // Аргументы isMin
  boolean isMin_; // Возвращаемое значение isMin
}
```

Перейдем к модификации программы. После выполнения первых трех шагов из раздела 2.2 процедура `calc` приобретает следующий вид:

```
Model d; // Объявление переменной модели

void calc() {
  int m = 0;
  for (int i = 1; i < a.length - 1; i++) {
    if (isMin(@a[i-1], @a[i], @a[i+1]) && @a[i] > m)
        m = @a[i];
  }
}
```

А процедура `isMin` осталась без изменений.

Выполнение шагов 4–6 изменяет обе процедуры:

```
void calc() {
  @m = 0;
  for (@i = 1; @i < @a.length-1; @i++) {
    if (isMin(@a[@i-1], @a[@i], @a[@i+1]) && @a[@i] > @m) @m =
        @a[@i];
  }
}

boolean isMin(int a, int b, int c) {
  @m = a > b ? a : b;
  return b == (@m > c ? @m : c);
}
```

Так как в рассматриваемой программе осуществляется единственный вызов процедуры, то на седьмом шаге программа не модифицируется.

После выполнения шагов восемь и девять имеем:

```
void calc() {
  @m = 0;
  for (@i = 1; @i < @a.length-1; @i++) {
    @isMin_a = @a[@i - 1];
    @isMin_b = @a[@i];
    @isMin_c = @a[@i + 1];
    isMin();
    if (@isMin_ && @a[@i] > @m)
        @m = @a[@i];
  }
}

boolean isMin(int a, int b, int c) {
  @m = a > b ? a : b;
  return b == (@m > c ? @m : c);
}
```

Модификация завершается применением шагов 10–12 к процедуре `isMin`. Итоговая программа выглядит следующим образом:

```
class Model {
  int[] a; // Глобальная переменная
  int calc_m, calc_i, isMin_m; // Локальные переменные
  int isMin_a, isMin_b, isMin_c; // Аргументы isMin
  boolean isMin_; // Возвращаемое значение isMin
}

Model d; // Объявление переменной модели

void calc() {
```

```
    @m = 0;
    for (@i = 1; @i < @a.length-1; @i++) {
      @isMin_a = @a[@i - 1];
      @isMin_b = @a[@i];
      @isMin_c = @a[@i + 1];
      isMin();
      if (@isMin_ && @a[@i] > @m)
           @m = @a[@i];
    }
  }

  void isMin() {
    @m = @a > @b ? @a : @b;
    @isMin_ = @b == (@m > @c ? @m : @c);
  }
```

Заметим, что при выделении модели, объем исходного кода несколько увеличился, но при автоматизированной обработке это несущественно.

3. Построение модели данных по рекурсивной программе

В отличие от итеративных программ, при рекурсивном вызове процедур, для каждой локальной переменной выделяется новая область памяти, в то время, как в модели ей соответствует только одна переменная. Таким образом, возникает проблема, связанная с тем, что при рекурсивном вызове могут изменяться локальные переменные другого экземпляра рекурсивной процедуры. Для учета этой особенности нужно несколько изменить правила изложенные в разделе 2.1.

3.1. Построение модели данных

Если в программе используется только косвенная рекурсия, то можно непосредственно применять правила именования, изложенные в разделе 2.1.

При использовании непосредственной рекурсии требуется ввести дополнительные переменные модели:

5. для каждого формального аргумента процедуры, в модель дополнительно к переменным, добавленным на шаге 3, добавляется переменная с именем вида

   <имя процедуры>_<имя формального аргумента>_

   Тип добавленной переменной соответствует типу формального аргумента.

Добавленные таким образом переменные будут использоваться как временные хранилища значений реальных аргументов вызываемой процедуры.

3.2. Модификация программы

Для хранения локальных переменных и реальных аргументов процедур требуется не только модель, но и дополнительная память. Для сохранения данных будет использован стек, как и при обычном выполнении.

Для этого введем оператор *квази-присваивания* (в программах он будет обозначаться “`@=`”) который не только изменяет значение своего левого аргумента, но и сохраняет замещенное значение в стеке. Так же нам понадобится оператор извлечения значений, сохраненных оператором *квази-присваивания*, обозначаемый “`?=`”. Эквиваленты этих операторов легко реализуются на любом языке программирования.

При модификации программ, использующих рекурсию, шаги 1-8 и 10-12 (раздел 2.2) остаются без изменений, а шаг 9 заменяется в зависимости от типа рекурсивного вызова.

Если вызов не является непосредственной рекурсией, то девятый шаг выглядит следующим образом (для сокращения записи в место *<имя процедуры>_<аргументM>* будем писать *имяM*):

- Вызовы процедур преобразуются следующим образом. Пусть вызывается процедура с *N* формальными аргументами. Обозначим имена этих аргументов следующим образом:

  аргумент1, *аргумент2*, ... *аргументN*.

  Реальные аргументы обозначим как

  выражение1, *выражение2*, ..., *выражениеN*

  соответственно. Тогда оператор вызова процедуры заменяется на:

  - *N* операторов квази-присваивания:

    `data.`*<имя процедуры>*`_`*<аргументM>* `@=` *<выражениеM>*`;`

    где *M* пробегает значения от 1 до *N*;
    - вызов процедуры без аргументов.
    - *N* операторов восстановления значений сохраненных квази-присваиваниями:

      `data.`*<имяM>* `?=;`

      где M пробегает значения от 1 до *N* в обратном порядке.

Если же вызов является непосредственный рекурсией, то шаг 9 еще более усложняется:

9. Вызовы процедур преобразуются следующим образом. Пусть вызывается процедура с *N* формальными аргументами. Обозначим имена этих аргументов следующим образом:

   аргумент1, *аргумент2*, ... *аргументN*.

   Реальные аргументы обозначим как

   выражение1, *выражение2*, ..., *выражениеN*

   соответственно. Тогда оператор вызова процедуры заменяется на:

   - *N* операторов присваивания вида

     `data.`*<имяM>*`_ =` *<выражениеM>*`;`

     где *M* пробегает значения от 1 до *N*.
   - *N* операторов квази-присваивания, копирующие значения реальных аргументов:

     `data.`*<имяM>* `@= data.`*<имяM>*`_;`

     где *M* пробегает значения от 1 до *N*.
   - Вызов процедуры без аргументов.
   - *N* операторов восстановления значений сохраненных квази-присваиваниями:

     `data.`*<имяM>* `?=;`

     где *M* пробегает значения от 1 до *N* в обратном порядке.

Указанные модификации позволяют формально выделять модель и преобразовывать программы использующие рекурсию.

3.3. Пример выделения модели и модификации программы

Рассмотрим изложенные модификации на примере программы, вычисляющей факториалы первых *n* натуральных чисел:

```
/** Количество чисел, для которых требуется вычислить
    факториал. */
int n;

/** Основная процедура. */
void calc() {
  for (int i = 1; i <= n; i++) {
    // Вывод значения факториала
    System.out.println(fact(i));
  }
}
```

```
/** Вычисляет факториал числа a. */
int fact(int a) {
  if (a == 0) {
    return 1;
  } else {
    return a * fact(a - 1);
  }
}
```

После выполнения шагов 1-8 имеем:

```
/** Модель данных. */
class Model {
  int n; // Глобальная переменная
  int calc_i; // Локальная переменная процедуры calc
  int fact_; // Возвращаемое значение процедуры calc
  // Переменная для формального параметра a процедуры fact
  int fact_a, fact_a_;
}
Model d;

void calc() {
  for (@i = 1; @i <= @n; @i++) {
    fact(@i);
    System.out.println(@fact_);
  }
}

int fact(int a) {
  if (@a == 0) {
    @fact_ = 1;
  } else {
    fact(@a - 1);
    @fact_ = @a * @fact_;
  }
}
```

После выполнения остальных шагов процедуры изменяется описание и вызовы процедуры `fact`:

```
void calc() {
  for (@i = 1; @i <= @n; @i++) {
    @fact_a @= @i;
    fact();
    @fact_a ?=;
    System.out.println(@fact_);
  }
}

void fact() {
  if (@a == 0) {
    @fact_ = 1;
  } else {
    @a_ = @a - 1;
    @a @= @a_;
    fact();
    @a ?=;
    @fact_ = @a * @fact_;
  }
}
```

Заметим, что вызов процедуры `fact` из `calc` не является прямой рекурсией, а вызова `fact` из себя самой ей является. По этому, код для первого из этих вызовов не использует переменную модели `@a_`.

Заключение

Предложенный метод позволяет производить выделение модели данных путем формального выполнения описанных шагов, что позволяет строить на его основе инструменты, осуществляющие выделение данных программы по ее исходному коду и модификацию исходной программы с целью использования построенной модели данных.

Литература

1. Шалыто А. А. SWITCH-технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука, 1998.
2. Туккель Н. И., Шалыто А. А., Шамгунов Н. Н. Реализация рекурсивных алгоритмов на основе автоматного подхода // Телекоммуникации и информатизация образования. — 2002. — № 5.
3. Корнеев Г.А., Шалыто А.А. Преобразование программ в систему взаимодействующих конечных автоматов. // Труды Второй Всероссийской Научной конференции «Методы и средства обработки информации». М.:МГУ.