Мартыненко Борис Константинович

УЧЕБНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ПРОЕКТ РЕАЛИЗАЦИИ АЛГОРИТМИЧЕСКИХ ЯЗЫКОВ: ПРИВЕДЕНИЯ

Аннотация

Рассматривается реализация приведений в объектно-ориентированной модели семантики на примере языка программирования Алгол 68.

Ключевые слова: априорный и апостериорный виды конструкций и значений, план приведений конструкции, приведение значений.

1. ВВЕДЕНИЕ

Напомним, что необходимость приведений возникает тогда, когда виды конкретных подконструкций, составляющих данную конструкцию, не удовлетворяют требованиям, определяемым грамматикой входного языка, по отношению к виду данной конструкции. Виды подконструкций, а также результатов их непосредственного исполнения, то есть значений, называются *априорными* видами. Чтобы значения подконструкций могли быть использованы конструкцией, они должны быть приведены к виду, определяемому вышеупомянутыми требованиями. Эти виды значений подконструкций после приведений называются *апостериорными*.

Априорные виды подконструкций устанавливаются по информации, извлекаемой из текста входной программы, а последовательности приведений их результатов определяются по грамматике входного языка, исходя из контекстных условий на соотношения видов подконструкций с учётом сортов позиций подконструкцией в данной конструкции.

В свою очередь, если данная конструкция сама является подконструкций некоторой объемлющей конструкции, то процесс установления приведений повторяется для объемлющей конструкции. Этот процесс заканчивается, когда очередная рассматриваемая конструкция является собственно программой.

2. СХЕМА РЕАЛИЗАЦИИ ПРИВЕДЕНИЙ

При проектировании приведений учитываются следующие соображения.

1. Выстраивание *плана приведений* для каждой конструкции – задача конвертера, а исполнение соответствующих действий над их значениями относится к времени счёта, то есть интерпретации семантического дерева входной программы.

2. План приведений – это последовательность действий по преобразованию *априорного* вида значения к требуемому *апостериорному* виду. Именно апостериорные значения подконструкций используются непосредственно при исполнении данной конструкции.

3. План приведений данной подконструкции формируется с учётом *сорта* позиции, занимаемой подконструкцией по отношению к объемлющей конструкции, а также имеющегося априорного и требуемого апостериорного вида этой подконструкции.

4. Описание языка Алгол 68 [2] различает пять сортов позиций[1], которые могут занимать подконструкции данной конструкции. Сорт позиции указывает, какие преобразования видов возможны в данной позиции, а также допустимый порядок их выполнения. Множество этих преобразований видов невелико – их всего шесть[2], однако их исполнение зависит от вида значений, над которыми они выполняются.

5. В рассматриваемой модели план приведений подконструкции данной конструкции представляется строкой (**`string`**), кодирующей последовательность приведений. Эти строки, относящиеся ко всем подконструкциям данной конструкции, передаются ей через её конструктор в момент создания и используются методом **`Run`** для приведений результатов подконструкций. Для этого метод **`Run`** вызывает полиморфную процедуру **`Coercing`** со строкой, планом приведений подконструкции, в качестве параметра.

6. Код плана приведений данной подконструкции интерпретируется процедурой **`Coercing`**, получающей априорное значение подконструкции через административную переменную **`UV`**.

7. По каждому символу этого кода интерпретатор вызывает соответствующий метод приведения полученного значения, выполняет его и выдаёт результат, то есть другой экземпляр объекта-значения, снова в **`UV`**.

8. Выполнение плана приведений подконструкции завершается, когда все символы кода использованы. В этот момент значение **`UV`** представляет апостериорное значение подконструкции. Оно используется методом **`Run`** конструкции, в соответствии с её семантикой.

Рассмотрим реализацию приведений на примере конструкции *присваивание*.

3. МОДЕЛИРОВАНИЕ ПРИСВАИВАНИЙ С УЧЁТОМ ПРИВЕДЕНИЙ

Заметим, что получатель присваивания – всегда имя, занимает мягкую позицию, а источник – сильную. Поэтому получатель можно разве лишь распроцедуривать[3], а источник приводить с учётом априорного вида источника и допустимых последовательностей приведений.

Определим дополнения к описанию родового типа объекта **`TAssignment`** в модуле **`CONSTRUCTS`** [5], связанных с реализацией приведений.

Во-первых, добавлены следующие поля данных:

1) **`DestinationCoercion`** – строка, представляющая последовательность приве-дений результата конструкции, играющей роль получателя присваивания.

2) **`SourceCoercion`** – строка, представляющая последовательность приведений результата конструкции, играющей роль источника присваивания.

Во-вторых, в конструктор **`Init`** объекта-конструкции присваивание конкретного вида вводятся два новых параметра:

1) **`dc`** – строка, инициализирующая поле данных **`DestinationCoercion`**.

2) **`sc`** – строка, инициализирующая поле данных ***`SourceCoercion`***.

---

[1] Сильная, крепкая, раскрытая, слабая, мягкая.

[2] Распроцедуривание, разыменование, объединение, обобщение, векторизация, опустошение.

[3] Это приведение относится к процедурам без параметров и не рассматривается до обсуждения процедур как значений в предстоящих публикациях.

В-третьих, поля данных **DestinationValue** и **SourceValue** в наследнике конкретного типа, скажем, **TBooleanAssignment**, являются указателями на значения получателя и источника, сначала априорных, а затем апостериорных видов.

Соответственно, метод **Run** конструкции *присваивание* дополняется вызовами вида **Coercing(DestinationCoercion)** и **Coercing(SourceCoercion)**, результаты которых дают апостериорные значения получателя и источника.

Поскольку план приведений есть последовательность элементарных приведений, относящихся к значениям, то их реализация должна описываться как методы объектов-значений соответствующих (априорных) видов.

Например, разыменование значений вида *reference to boolean* описывается как метод **deref** в объекте-значении типа **Tref_bool**, а разыменование значений вида *reference to reference to Boolean* описывается как метод **deref** в объекте-значении типа **Tref_ref_bool** в модуле **PLAIN_VALUES**.

Таким образом, **deref** – полиморфный метод, применимый к именам любых видов при условии, что существуют описания их представлений на инструментальном языке.

Другие методы, реализующие приведения, тоже относятся к избранным видам значений. Например, *обобщение*, может быть преобразованием целого в вещественное (см. пример 2), или вещественного в комплексное, или массива логических в битовое, или массива литерных в байтовое значение.

Описание реализации конструкции *присваивание* вида *reference to* MODE, где MODE обозначает любой вид, следует приведенному образцу:

```
{ Присваивание вида reference to MODE }
  constructor TMODEAssignment.Init (r : PChar; D, S : PConstruct;
                                    da, sa : PAddr; dc, sc : string);
  begin Representation := r;
        Destination := D;           Source := S;
        DestinationAddr := da;      SourceAddr := sa;
        DestinationCoercion := dc; SourceCoercion := sc;
  end;
  function TMODEAssignment.Show : PChar;
  var Bf, Bf1 : array [0..512] of char;
  begin if Destination <> nil
        then StrPCopy (Bf, Destination^.Show)
        else StrPCopy (Bf, DestinationAddr^.Show);
        { Destination представлен в виде PChar }
        if Source <> nil
        then begin StrPCopy (Bf1, Representation);
                   StrCat (Bf1, Source^.Show)
             end
        else begin StrPCopy (Bf1, Representation);
                   StrCat (Bf1, SourceAddr^.Show)
             end;
         { Source представлен в виде PChar }
       StrCat (Bf, Bf1);
    Show := Bf
  end;

procedure TMODEAssignment.Run;
var x : Pref_mode;
begin
{$IFDEF diag}
  writeln ('В ПОСЛЕДУЮЩЕМ ПРИСВАИВАНИИ ВИДА reference to MODE:');
{$ENDIF}
```

```
if Destination  <> nil
then { Априорное значение получателя доставляется конструкцией }
begin
  Destination^.Run;
  DestinationValue := Pref_mode (UV) { Априорное значение получателя }
end
else if DestinationAddr <> nil
     then { Получатель - идентификатор в стеке }
     begin DestinationValue := Pref_mode (DestinationAddr^.GetValue);
          { Априорное значение получателя }
     end
     else Haltx (10);
if Source <> nil
then { Априорное значение источника доставляется конструкцией }
begin
  Source^.Run;
  SourceValue := PMODEValue (UV) { Априорное значение источника }
end
else if SourceAddr <> nil
     then { Источник - идентификатор в стеке }
       SourceValue := PMODEValue (SourceAddr^.GetValue);
       { Априорное значение источника }
     else Haltx (10);
if DestinationCoercion <> ''
then begin
       UV := DestinationValue;
       coercing (DestinationCoercion);
       DestinationValue := Pref_mode (UV)
     end
else {$IFDEF diag}
      writeln ('априорное значение получателя = ',
              DestinationValue^.Show, ' не требует приведений');
      {$ENDIF};
if SourceCoercion <> ''
then begin
       {$IFDEF diag} writeln ('априорное значение источника = ',
                              SourceValue^.Show);  {$ENDIF}
       coercing (SourceCoercion);
       SourceValue := PBooleanValue (UV);
       {$IFDEF diag} writeln ('апостериорное значение источника = ',
                             SourceValue^.Show);  {$ENDIF}
     end
else  {$IFDEF diag} writeln ('априорное значение источника = ',
       SourceValue^.Show, ' не требует приведений');{$ENDIF};

if DestinationValue^.Scope <= SourceValue^.Scope
then
begin  x := New (Pref_mode, Init (nil));
       x^.Value := SourceValue;
       DestinationAddr^.PutValue (x);
       UV := DestinationAddr^.GetValue;
end
else {область действия получателя старше области действия источника}
     Haltx (1);
end;
```

Здесь вхождения **MODE** и **mode** согласуются по таблице 1.

Табл. 1

Вид присваивания	`MODE`	`mode`
reference to boolean	`boolean`	`bool`
reference to integral	`integral`	`int`
reference to real	`real`	`real`
reference to reference to boolean	`reference_to_ boolean_`	`ref_bool`
reference to reference to integral	`reference_to_integral_`	`ref_int`
reference to reference to real	`reference_to_real_`	`ref_real`
...	...	...

Мнемоника обозначений типов конструкций и значений основана не на виде получателя, а на виде источника данного присваивания[1].

Например, если речь идёт о присваивании вида *reference to boolean*, то апостериорный вид источника есть *boolean*; в присваивании вида *reference to reference to boolean* источник вида *reference to boolean* и т. д.

Присваивание вида *reference to reference to* MODE использует косвенное имя в качестве значения получателя и, может быть, источника, если MODE обозначает вид имени.

В рассматриваемой модели семантики входного языка следует уточнить представление значений косвенных имён, поскольку это не было сделано ранее.

4. КОСВЕННЫЕ ИМЕНА

Напомним, что в Алголе 68 имена – это значения, которые используются для именования или ссылок на другие значения. В этом состоит их предназначение в семантике языка. Отношение именования между двумя значениями устанавливается в результате исполнения конструкции *присваивание*, при этом значение получателя замещается значением источника.

Соответствие видов значений, между которыми устанавливается отношение именования, должно быть тиким, как показано в следующей мнемонической формуле:

(1) *reference to* MODE := MODE,

где MODE обозначает один и тот же вид.

Если значение, на которое ссылается данное имя, само является именем, то данное имя называется косвенным.

Формально значение вида *reference to* MODE, где MODE само является именем вида *reference to* MODE_1, называется *косвенным именем*, независимо от того, какой вид обозначает MODE_1.

Среди имён любого вида существует особое значение `nil`, которое не именует никакого другого значения. Такое специальное имя нельзя использовать в качестве значения получателя в присваивании, а также *разыменовывать*, то есть получать значение, именуемое этим именем.

Например, пусть идентификатор *rb* обладает именем, которое может именовать любое логическое значение. Если это имя не `nil`, то ему можно присвоить, скажем, значение `false`. Для этого достаточно исполнить присваивание

(2) *rb* := `false`.

В (2) значения получателя и источника удовлетворяют соотношению (1) при MODE = *boolean,* и потому в результате исполнения присваивания (2) между именем,

[1] Ради сокращения обозначений.

которым обладает идентификатор *rb*, в качестве получателя, и значением **`false`**, в качестве результата исполнения конструкции[1] источника, устанавливается отношение именования.

В модельном представлении значений, о которых идёт речь, эти связи именования представлены на рис. 1.

В случае, когда имя *rb* равно **`nil`**, между ним и каким-то ещё логическим значением отношение именования установить невозможно. Эта ситуация представлена на рис. 2.

Напомним, что значения любых видов в рассматриваемой модели семантики представляются как объекты-контейнеры, доступ к которым исполняемой программы производится через типизированные указатели, соответствующие виду значения. Собственно значение, содержащееся в объекте-контейнере, представляется полем **`Value`**, тоже типизированным.

Принимая во внимание вышесказанное, схематически проиллюстрируем связи между именами вида *reference to reference to boolean* и *reference to boolean*, а также между значениями вида *reference to boolean* и вида *boolean* (см. рис. 3).

Здесь *rrb* – косвенное имя (первого порядка), а значение, именуемое им, тоже имя, но не косвенное. На рисунках 1–3 рамки изображают объекты-контейнеры значений.

Как показывает опыт, содержательное использование косвенных имён, порядка больше трёх, трудно вообразить.

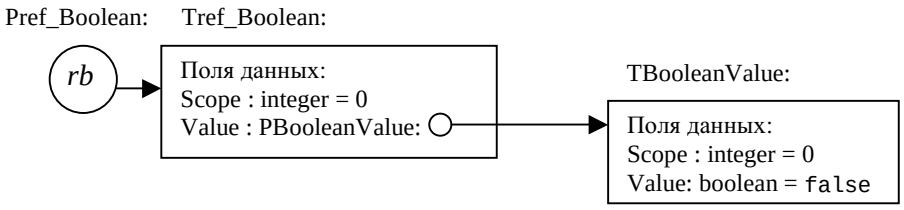

Рис. 1. Связи между именем *rb* вида *reference to boolean* и значением **`false`** вида *boolean*

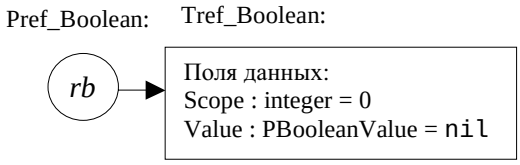

Рис. 2. Имя *rb* вида *reference to boolean*, равное **`nil`**, не может именовать ничего

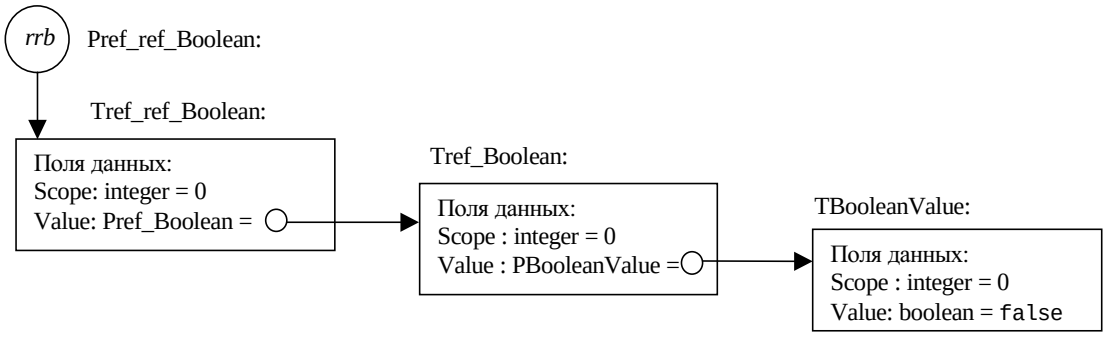

Рис. 3. Представление косвенного имени первого порядка

[1] Изображение логического **`false`**.

Обратимся ещё раз к ситуации, показанной на рис. 2. В ней мы имеем дело со значением вида *reference to boolean*, то есть с именем, которому не присвоено никакого значения, но известно, какого вида оно должно быть.

Вопрос: существует ли возможность в языке Алгол 68 создать эту ситуацию?

Ответ: Да. Такая ситуация создаётся при исполнении программы

```
ref bool rb = nil;  rb := false
```

Действительно, первая конструкция этой программы – описание тождества. Источник этого описания тождества есть конструкция *псевдоимя* (*nihil*), представленная в тексте символом **nil**, исполнение которой даёт имя вида *reference to boolean*, которое ничего не именует.

Вторая конструкция программы, *присваивание*, получатель которой есть использующее вхождение идентификатора *rb*, а источник – изображение логического, представленный в тексте символом **false**. До присваивания идентификатор *rb* обладает псевдоименем вида *reference to boolean*, а после – исполнение этого присваивания имя, которым обладает идентификатор *rb*, начинает именовать логическое значение *false*.

Наша задача – воспроизвести подобный эффект в описываемой модели семантики.

5. ЗНАЧЕНИЯ ПО УМОЛЧАНИЮ ПРОТИВ ПСЕВДОИМЁН

Имена, как известно, образуются в результате исполнения генераторов, обсуждавшихся в [5], и псевдоимён, упомянутых выше. Последние из названных представляются как объекты-контейнеры обычных имен, но поля **Value** которых равны **nil**, а не обычному указателю на значение указанного типа. Благодаря этому, в модельном представлении семантики Алгола 68, для получения значения псевдоимени (nihil) предлагается использовать конструкцию генератора соответствующего вида, а затем перекрывать значение образца значением **nil**. Такая реализация псевдоимён с одной стороны не позволяет разыменовывать псевдоимена, а с другой – даёт возможность замещать значение поля **Value**, равное **nil**, значением источника присваивания соответствующего вида.

Например, генератор логического, выдающий псевдоимя, имеет следующее описание метода **Run**:

```
{ Генератор логического }
procedure TBooleanGenerator.Run;
var ref_to_bool : Pref_bool;
begin
  { Создание нового экземпляра имени логического значения }
    ref_to_bool^.Value := New (PBooleanValue, Init (false));
  { Замещение значения поля Value в имени логического указателем nil }
    ref_to_bool^.Value := nil;
  { Обновление области действия имени логического }
    if is_loc
    then { генератор локальный }
           ref_to_bool^.Scope := CurrentLevel
    else { генератор глобальный }
           ref_to_bool^.Scope := 0;
  { Выдача пседоимени логического в UV }
    UV := ref_to_bool;
end;
```

Другой вариант семантики имён, часто используемый в языках программирования, вместо понятия псевдоимени Алгола 68, это значения, присваиваемые именам по умолчанию (*default value*) в момент их создания.

Поскольку в Алголе 68 имена создаются генераторами, то метод **Run** генератора создаёт значение вида, специфицированного фактическим описателем генератора, и выдаёт имя, ссылающееся на него.

Так как в [5] генераторы косвенных имён не рассматривались, то самое время сделать это сейчас.

6. ГЕНЕРАТОРЫ КОСВЕННЫХ ИМЕН

Напомним, что родовой тип генератора любого вида определяется предописанием

```
PGenerator = ^TGenerator;
TGenerator = object (TConstruct)
 (* Наследуемые поля данных:
   Representation : PChar; *)
 is_loc : Boolean;
 constructor Init (r : PChar; loc : Boolean);
 function Show : PChar; virtual;
 procedure Run; virtual;
end;
```

с реализацией методов

```
constructor TGenerator.Init (r : PChar; loc : Boolean);
begin Representation := r; {Представаление генератора во входной программе}
      is_loc := loc {если is_loc = true, то генератор локальный,
                     если is_loc = false, то генератор глобальный}
end;
function TGenerator.Show : PChar; begin abstract end;
procedure TGenerator.Run; begin abstract end;
```

Генератор (не косвенного) имени конкретного вида '**reference to MODE**', где **MODE** не начинается с префикса '**reference to**', предописывается по образцу:

```
PMODEGenerator = ^TMODEGenerator;
TMODEGenerator = object (TGenerator)
constructor Init (r : PChar; loc : Boolean);
function Show : PChar; virtual;
procedure Run; virtual;
end;
```

с реализацией методов

```
constructor TMODEGenerator.Init (r : PChar; loc : Boolean);
begin inherited Init (r, loc) end;
function TMODE.Show : PChar;
begin Show := Representation end;
procedure TMODEGenerator.Run;
var v : PMODEValue;
    ref_to_mode : Pref_MODE;
begin { Создание контейнера значения вида MODE }
    v := New (PMODEValue, Init (0));
    { Создание нового экземпляра имени значения вида MODE }
    ref_to_mode := New (Pref_MODE, Init(v));
    { Обновление области действия имени }
    if is_loc
    then ref_to_mode^.Scope := CurrentLevel
    else ref_to_mode^.Scope := 0;
    { Выдача имени вида reference to MODE как указатель
      на контейнер значения типа mode }
    UV := ref_to_mode;
end;
```

Здесь вхождения **MODE** и **mode** в вышеприведённом образце согласуются так же, как определено таблицей 1 (см. п. 3).

Генераторы косвенных имён описываются аналогично, лишь с той разницей, что разделы описаний переменных и их создание в методах **Run** пишутся по образцам, в зависимости от порядка косвенности.

Именно, описания косвенных имён порядка 1 следуют образцу:

```
var v : PMODEValue;
    ref_to_mode : Pref_MODE;
    ref_to_ref_to_mode : Pref_ref_MODE;
```

а создание новых экземпляров этих переменных образцу:

```
ref_to_mode := New (Pref_mode, Init (v));
ref_to_ref_to_mode := New (Pref_ref_MODE, Init(ref_to_mode));
```

описания косвенных имён порядка 2 следуют образцу:

```
var v : PMODEValue;
    ref_to_mode : Pref_MODE;
    ref_to_ref_to_mode : Pref_ref_MODE;
    ref_to_ref_to_ref_to_mode : Pref_ref_ref_MODE;
```

а создание новых экземпляров этих переменных следуют образцу:

```
ref_to_mode := New (Pref_mode, Init (v));
ref_to_ref_to_mode := New (Pref_ref_MODE, Init(ref_to_mode));
ref_to_ref_to_ref_to_mode := New(Pref_ref_ref_MODE, Init(ref_ref_to_mode));
```

... и т. д.

Здесь **v**, **ref_to_mode**, **ref_to_ref_to_mode**, **ref_to_ref_to_ref_to_mode** и т. д. – указатели на контейнеры, представляющие соответственно значения видов

MODE (не имя),

reference to MODE,

reference to reference to MODE,

reference to reference to reference to MODE, и т. д.

Например, реализация генератора по первому образцу при MODE = *boolean* даёт косвенное имя порядка 1, представленное на рис. 3.

7. ИНТЕРПРЕТАТОР ПРИВЕДЕНИЙ

Код плана приведений конструкции, переданный объекту-конструкции через её конструктор, интерпретируется специальной процедурой **Coercing**, которая играет роль диспетчера. Каждая литера кода расшифровывается как вызов метода, осуществляющего соответствующее приведение значения, находящегося в **UV**. Приведённое значение остаётся в **UV**. После выполнения всей последовательности приведений, заданной данным кодом, в **UV** остаётся апостериорное значение, которое и используется конструкцией, инициализировавшей эти приведения.

Процедура **Coercing** описывается в модуле **COERCION** следующим образом:

```
Unit COERCION;
{$define dial }
{ Все методы, реализующие приведения, получают априорное значение в UV и
  выдают апостериорное значение на выходе тоже в UV }
interface
uses objects, Strings,
     VALUES, PLAIN_VALUES, ENVIRON, AUXILIARY, REF_PLAIN;
procedure Coercing (cc : string);
```

```
implementation
 { Интерпретатор приведений }
 procedure Coercing (cc : string);
  { Приведения выполняются на априорным результатом конструкции в UV.
    Апостериорное значение остаётся в UV. }
 var i : integer;
 begin
  for i := 1 to length (cc) do
  if cc [i] = 'a' then {разыменование}
                  begin UV^.deref;
                     {$IFDEF dial} writeln ('Выполнено dereferencing');
{$ENDIF} end else
  if cc [i] = 'b' then {распроцедуривание} else
  if cc [i] = 'c' then {объединение} else
  if cc [i] = 'd' then {обобщение целого до вещественного}
                  begin UV^.widening;
          {$IFDEF dial} writeln ('Выполнено widening') {$ENDIF} end else
  if cc [i] = 'e' then {обобщение вещественного до комплексного} else
  if cc [i] = 'f' then {обобщение массива логических до битового} else
  if cc [i] = 'g' then {обобщение массива литерных до байтового} else
  if cc [i] = 'i' then {векторизация скалярного до одномерного массива}
else
  if cc [i] = 'j' then {векторизация имени скалярного
                        до одномерного массива имён} else
  if cc [i] = 'l' then {векторизация массива до массива
                        на 1 большей размерности}
                  else
  if cc [i] = 'm' then {векторизация имени массива до массива имён
                        на 1 большей размерности} else
  if cc [i] = 'o' then {опустошение прямое} else
  if cc [i] = 'p' then {опустошение после раскрытия};
         {$IFDEF dial} writeln ('UV = ', UV^.Show);{$ENDIF}
 end;
end.
```

Фактически в ней задействованы только два приведения: разыменование, адресуемое к любым именам, и обобщение целого до вещественного. Остальные приведения представлены только комментариями до будущих обсуждений.

8. ИЗМЕНЕНИЯ ОПИСАНИЯ ЗНАЧЕНИЙ В СВЯЗИ С ПРИВЕДЕНИЯМИ

Во-первых, модуль **VALUES**, описывающий родовой тип объектов-значений, дополняется абстрактными мет**Coercing**одами всех шести приведений.

```
Unit VALUES;
{ Реализация объектов-значений }
interface
uses objects, Strings;
type
  { Абстрактный объект - родовой тип для значений любых видов }
     PValue = ^TValue;
     TValue = object (TObject)
      Scope : integer;
      constructor Init (level : integer);
      function Show : PChar; virtual;
      function GetScope : integer;
```

```
      procedure deref{erencing}; virtual;
      procedure widening; virtual;
      procedure deproc{during}; virtual;
      procedure uniting; virtual;
      procedure rowing; virtual;
      procedure voiding; virtual;
    end;
  implementation
  constructor TValue.Init (level : integer);
  begin Scope := level end;
  function TValue.GetScope : integer;
  begin GetScope := Scope end;
  function TValue.Show : PChar;
  begin abstract end;
  procedure TValue.deref{erencing};
  begin abstract end;
  procedure TValue.widening;
  begin abstract end;
  procedure TValue.deproc{during};
  begin abstract end;
  procedure TValue.uniting;
  begin abstract end;
  procedure TValue.rowing;
  begin abstract end;
  procedure TValue.voiding;
  begin abstract end;
end.
```

Во-вторых, модули, определяющие значения конкретных видов, пополняются методами, заменяющими соответствующие абстрактные.

Например,

```
Unit REF_PLAIN;
{ Представление имён простых видов }
interface
uses objects, Strings,
     VALUES, PLAIN_VALUES, ENVIRON, AUXILIARY;
type
    { Имена вида reference to Boolean }
     Pref_bool = ^Tref_bool;
     Tref_bool = object (TValue)
     { Scope - унаследованное поле }
      Value : PBooleanValue;
      constructor Init (v : PBooleanValue);
      destructor Done; virtual;
      function Show : PChar; virtual;
      procedure deref; virtual;
     end;
   { Имена вида reference to reference to boolean }
     Pref_ref_bool = ^Tref_ref_bool;
     Tref_ref_bool = object (TValue)
     { Scope - унаследованное поле }
      Value : Pref_bool;
      constructor Init (v : Pref_bool);
      destructor Done; virtual;
      function Show : PChar; virtual;
      procedure deref; virtual;
     end;
```

```
    { Имена вида reference to integral }
   Pref_int = ^Tref_int;
   Tref_int = object (TValue)
   { Scope - унаследованное поле }
    Value : PIntegralValue;
    constructor Init (v : PIntegralValue);
    destructor Done; virtual;
    function Show : PChar; virtual;
    procedure deref; virtual;
   end;
  { Имена вида reference to reference to integral }
   Pref_ref_int = ^Tref_ref_int;
   Tref_ref_int = object (TValue)
   { Scope - унаследованное поле }
    Value : Pref_int;
    constructor Init (v : Pref_int);
    destructor Done; virtual;
    function Show : PChar; virtual;
    procedure deref; virtual;
   end;
  { Имена вида reference to real }
   Pref_real = ^Tref_real;
   Tref_real = object (TValue)
   { Scope - унаследованное поле }
    Value : PRealValue;
    constructor Init (v : PRealValue);
    destructor Done; virtual;
    function Show : PChar; virtual;
    procedure deref; virtual;
   end;
  { Имена вида reference to reference to real }
   Pref_ref_real = ^Tref_ref_real;
   Tref_ref_real = object (TValue)
   { Scope - унаследованное поле }
    Value : Pref_real;
    constructor Init (v : Pref_real);
    destructor Done; virtual;
    function Show : PChar; virtual;
    procedure deref; virtual;
   end;
implementation
 { Имена вида reference to boolean }
 constructor Tref_bool.Init (v : PBooleanValue);
 begin inherited Init (0); Value := v end;
 function Tref_bool.Show : PChar;
 var Bf, Bf1: array [0..512] of char;
     UVsafe : Pref_bool;
     v : PBooleanValue;
 begin v := New (PBooleanValue, Init (false));
       UVsafe:= New (Pref_bool, Init (v));
       UVsafe := Pref_bool (UV);
       Bf := "ref-> "; self.deref;
       Bf1 := UV^.Show; StrCat (Bf, Bf1); Show := Bf;
       UV := UVsafe
 end;
 procedure Tref_bool.deref;
 begin if Value <> nil then UV := Value else Haltx (2) end;
```

```
 { Имена вида reference to reference to Boolean }
 constructor Tref_ref_bool.Init (v : Pref_bool);
 begin inherited Init (0); Value := v end;
 function Tref_ref_bool.Show : PChar;
 var Bf, Bf1: array [0..512] of char;
     UVsafe : Pref_ref_bool;
     UVsafe1 : Pref_bool;
     v : PBooleanValue;
 begin
       v := New (PBooleanValue, Init (false));
       UVsafe1 := New (Pref_bool, Init (v));
       UVsafe := New (Pref_ref_bool, Init (UVsafe1));
       UVsafe := Pref_ref_bool (UV);
       Bf := "ref->ref-> "; self.deref; UV^.deref;
       Bf1 := UV^.Show; StrCat (Bf, Bf1); Show := Bf;
      UV := UVsafe
 end;
 procedure Tref_ref_bool.deref;
 { Эта процедура портит UV }
 begin if Value <> nil then UV := Value else Haltx (2) end;
 { Имена вида reference to integral }
 constructor Tref_int.Init (v : PIntegralValue);
 begin inherited Init (0); Value := v end;
 function Tref_int.Show : PChar;
 var Bf, Bf1: array [0..512] of char;
 begin Bf := 'ref-> '; self.deref;
       Bf1 := UV^.Show; StrCat (Bf, Bf1); Show := Bf
end;
procedure Tref_int.deref;
begin if Value <> nil then UV := Value else Haltx (2) end;
 { Имена вида reference to reference to integral }
constructor Tref_ref_int.Init (v : Pref_int);
begin inherited Init (0); Value := v end;
function Tref_ref_int.Show : PChar;
var Bf, Bf1: array [0..512] of char;
    UVsafe : Pref_ref_int;
    UVsafe1 : Pref_int;
    v : PIntegralValue;
begin
      v := New ( PIntegralValue, Init (0));
      UVsafe1 := New (Pref_int, Init (v));
      UVsafe := New (Pref_ref_int, Init (UVsafe1));
      UVsafe := Pref_ref_int (UV);
      Bf := 'ref->ref-> '; self.deref; UV^.deref;
      Bf1 := UV^.Show; StrCat (Bf, Bf1); Show := Bf;
      UV := UVsafe
end;
procedure Tref_ref_int.deref;
begin if Value <> nil then UV := Value else Haltx (2) end;
   { Имена вида reference to real }
constructor Tref_real.Init (v : PRealValue);
begin inherited Init (0); Value := v end;
function Tref_real.Show : PChar;
var Bf, Bf1: array [0..512] of char;
```

```
      UVsafe : Pref_real;
      v : PRealValue;
 begin v := New (PRealValue, Init (0));
         UVsafe:= New (Pref_real, Init (v));
         UVsafe := Pref_real (UV);
         Bf := 'ref-> '; self.deref;
         Bf1 := UV^.Show; StrCat (Bf, Bf1); Show := Bf;
         UV := UVsafe
 end;
 procedure Tref_real.deref;
 begin if Value <> nil then UV := Value else Haltx (2) end;
  { Имена вида reference to reference to real }
 constructor Tref_ref_real.Init (v : Pref_real);
 begin inherited Init (0); Value := v end;
 function Tref_ref_real.Show : PChar;
 var Bf, Bf1: array [0..512] of char;
 begin Bf := 'ref->ref-> '; self.deref; UV^.deref;
       Bf1 := UV^.Show; StrCat (Bf, Bf1); Show := Bf
 end;
 procedure Tref_ref_real.deref;
 begin if Value <> nil then UV := Value else Haltx (2) end;
```

Замечание. В этом модуле полиморфный метод **Show** вызывает метод **deref**, который портит **UV**. Поэтому в начале метод **Show** сохраняется текущее значение **UV** во вспомогательной переменной **UVsafe** с надлежащей её инициализацией, которое восстанавливается перед выходом из метода **Show**.

Кроме того, в модуль **PLAIN_VALUES** добавлен метод **widening** обобщения целого до вещественного.

```
procedure TIntegralValue.widening;
  { Обобщение целого до вещественного }
var i : PIntegralValue;
    r : PRealValue;
    v : integer;
begin i := PIntegralValue (UV);
      v := i^.Value;
      r := New (PRealValue, Init (v));
      UV :=r;
end;
```

Наконец, приведём вспомогательный модуль, содержащий процедуру **Haltx** диагностирования ошибок периода исполнения входной программы пользователя.

```
unit AUXILIARY;
interface
uses CRT;
procedure Haltx (i : integer);
implementation
procedure Haltx (i : integer);
begin write ('   Error ', i, ': ');
     case i of 1: write ('Нарушение областей действия в присваивании');
               2: write ('Разыменование псевдоимени');
                  {...}
          else  write ('Неизвестная ошибка!');
      end; readln;
       Halt (i)
end;
end.
```

Теперь всё готово для демонстрации двух примеров использования косвенных имён в конструкциях, а именно в описаниях тождества, генераторах и присваиваниях.

9. ПРИМЕР 1: НЕКОСВЕННОЕ ИМЯ ЛОГИЧЕСКОГО

В [5] рассматривался автономный тест **Name_test1** на использования косвенных имён и разыменование. Тест автономный в том смысле, что в нём описывается процесс создания и разыменования значений указанного вида непосредственно, без использования конструкций реализуемого языка.

Здесь приводится настоящий тест программы на основе логических значений и имён.

Листинг I. Построение семантического дерева программы

```
.begin .bool b = .true; .ref.bool rb = .loc.bool; rb := b;
       .ref.ref.bool rrb = .loc.ref.bool; rrb := rb; rb := rrb
.end
```

и её исполнения

```
program eq_dec_test_1;
uses CRT, objects, Strings,
     VALUES, PLAIN_VALUES, STANDART, CONSTRUCTS, CLAUSES, ENVIRON,
     DECLARATIONS, COERCION;
var bds, bool_default, ref_to_bool_default, r : string;
    bd : PBooleanDenotation;
    b, rb, rrb : PTag;
    TagList1, TagList2, TagList3 : PTagList;
    bg : PBooleanGenerator;
    bg1 : PReference_to_boolean_Generator;
    IdentityDeclaration1,
    IdentityDeclaration2,
    IdentityDeclaration3 : PIdentityDeclaration;
    Addr00, Addr01, Addr02 : PAddr;
    Assignment : PBooleanAssignment;
    Assignment1 : PReference_to_boolean_Assignment;
    Assignment2 : PBooleanAssignment;
    cl0 : PConstructList;
    Range0 : PRange;
    main : PClosedClause;
begin ClrScr;
 writeln ('Тестирование программы Алгола 68:');
 writeln ('.begin .bool b = .true; .ref.bool rb = .loc.bool; rb := b;');
 writeln ('.ref.ref.bool rrb = .loc.ref.bool; rrb := rb; rb := rrb .end');
 writeln ('ПРОСТРАНСТВО ДАННЫХ:'#10#13);
 { СОЗДАНИЕ СТЕКА ДАННЫХ }
   Stack := New (PStack, Init (1));
   writeln ('Стек на ', Stack^.Limit, ' участок');
 { СОЗДАНИЕ ТАБЛИЦЫ DISPLAY }
   Display := New (PDisplay, Init (1));
   writeln ('Display на ', Display^.Limit,' участок'#13#10);
   writeln ('=== СОЗДАНИЕ СЕМАНТИЧЕСКОГО ДЕРЕВА ПРОГРАММЫ ===');
 { СОЗДАНИЕ КОНСТРУКЦИЙ БЛОКА 0 }
 bool_default := '.loc.bool';
 bds := 'true';
 ref_to_bool_default:= '.loc.ref.bool';
 bd := New (PBooleanDenotation, Init (@bds[1], true));
 writeln ('Изображение логического: ', bd^.Show);
 writeln ('Создание Tag''a b ');
 b := New (PTag, Init (true, 'b', bd));
 writeln ('Tag: ', b^.Show);
 bg := New (PBooleanGenerator, Init (@bool_default[1], true));
 writeln ('Генератор логического: ', bg^.Show);
 writeln ('Создание Tag''a rb ');
 rb := New (PTag, Init (true, 'rb', bg));
 writeln ('Tag: ', rb^.Show);
```

```
 bg1 := New (PReference_to_boolean_Generator,
             Init (@ref_to_bool_default[1], true));
 writeln ('Создание Tag''a rrb ');
 rrb := New (PTag, Init (true, 'rrb', bg1));
 writeln ('Tag: ', rb^.Show);
 TagList1 := New (PTagList, Init (1, 0));
 TagList1^.Insert (b);
 writeln ('Список Tag''ов создан: ', TagList1^.Show);
 TagList2 := New (PTagList, Init (1, 0));
 TagList2^.Insert (rb);
 writeln ('Список Tag''ов создан: ', TagList2^.Show);
 TagList3 := New (PTagList, Init (1, 0));
 TagList3^.Insert (rrb);
 writeln ('Список Tag''ов создан: ', TagList3^.Show);
 IdentityDeclaration1 := New (PIdentityDeclaration,
                              Init ('.bool', TagList1));
 write ('Конструкция описание тождества создана: ');
 writeln (IdentityDeclaration1^.Show);
 IdentityDeclaration2 := New (PIdentityDeclaration,
                              Init ('.ref.bool', TagList2));
 write ("Конструкция описание тождества создана: ');
 writeln (IdentityDeclaration2^.Show);
 IdentityDeclaration3 := New (PIdentityDeclaration,
                              Init ('.ref.ref.bool', TagList3));
 write ('Конструкция описание тождества создана: ');
 writeln (IdentityDeclaration3^.Show);
 Addr00 := New (PAddr, Init (0, 0));
 writeln (#13#10'Создан адрес Addr00: ', Addr00^.Show);
 Addr01 := New (PAddr, Init (0, 1));
 writeln ('Создан адрес Addr01: ', Addr01^.Show);
 Addr02 := New (PAddr, Init (0, 2));
 writeln ('   Создан адрес Addr02: ', Addr02^.Show);
 { Создание конструкции присваивание: rb := b }
 r := ' := ';
 Assignment := New (PBooleanAssignment,
                    Init (@r[1], nil, nil, Addr01, Addr00, nil, nil));
 writeln (#13#10'  Конструкция присваиавние создана: ', Assignment^.Show);
 { Создание конструкции присваивание: rrb := rb }
 r := ' := ';
 Assignment1 := New (PReference_to_boolean_Assignment,
                     Init (@r[1], nil, nil, Addr02, Addr01, nil, nil));
 writeln (#13#10'  Конструкция присваиавние создана: ', Assignment1^.Show);
 Assignment2 := New (PBooleanAssignment,
                Init (@r[1], nil, nil, Addr01, Addr02, nil, coercing));
 { Создание списка конструкций блока 0 }
 cl0 := New (PConstructList, Init (6, 0));
 cl0^.Insert (IdentityDeclaration1);
 cl0^.Insert (IdentityDeclaration2);
 cl0^.Insert (Assignment);
 cl0^.Insert (IdentityDeclaration3);
 cl0^.Insert (Assignment1);
 cl0^.Insert (Assignment2);
 Range0 := New (PRange, Init (0, 10, cl0));
 main := New (PClosedClause, Init (Range0));
 writeln ('СЕМАНТИЧЕСКОЕ ДЕРЕВО ПРОГРАММЫ СОЗДАНО:', main^.Show);
 writeln ('ЗАПУСК ПРОГРАММЫ ... '#13#10);
 main^.Run;
 writeln ('ПРОГРАММА ВЫПОЛНЕНА!'); readln;
end.
```

```
 Тестирование программы Алгола 68:
 .begin .bool b = .true; .ref.bool rb = .loc.bool; rb := b;
 .ref.ref.bool rrb = .loc.ref.bool; rrb := rb; rb := rrb .end

ПРОСТРАНСТВО ДАННЫХ:

Стек на 1 участок
Display на 1 участок

======= СОЗДАНИЕ СЕМАНТИЧЕСКОГО ДЕРЕВА ПРОГРАММЫ =======

Изображение логического: true
Создание Tag'а b
Tag: b = true
Генератор логического: .loc.bool
Создание Tag'а rb
Tag: rb = .loc.bool
Создание Tag'а rrb
Tag: rb = .loc.bool
Список Tag'ов создан: b = true
Список Tag'ов создан: rb = .loc.bool
Список Tag'ов создан: rrb = .loc.ref.bool

Конструкция описание тождества создана: .bool b = true
Конструкция описание тождества создана: .ref.bool rb = .loc.bool
Конструкция описание тождества создана: .ref.ref.bool rrb = .loc.ref.bool

Создан адрес Addr00: < 0, 0 >
Создан адрес Addr01: < 0, 1 >
Создан адрес Addr02: < 0, 2 >

Конструкция присваивние создана: < 0, 1 > := < 0, 0 >

Конструкция присваивние создана: < 0, 2 > := < 0, 1 >

СЕМАНТИЧЕСКОЕ ДЕРЕВО ПРОГРАММЫ СОЗДАНО:
BEGIN(0)
  [0] .bool b = true
  [1] .ref.bool rb = .loc.bool
  [2] < 0, 1 > := < 0, 0 >
  [3] .ref.ref.bool rrb = .loc.ref.bool
  [4] < 0, 2 > := < 0, 1 >
  [5] < 0, 1 > := < 0, 2 >
END(0)

ЗАПУСК ПРОГРАММЫ ...

ConstructList:
  [0] .bool b = true
  [1] .ref.bool rb = .loc.bool
  [2] < 0, 1 > := < 0, 0 >
  [3] .ref.ref.bool rrb = .loc.ref.bool
  [4] < 0, 2 > := < 0, 1 >
  [5] < 0, 1 > := < 0, 2 >

Состояние стека после исполнения .bool b = true:

Display [0] ::
[0] b => true

Состояние стека после исполнения .ref.bool rb = .loc.bool:

Display [0] ::
[0] b => true
[1] rb => ref-> false

В ПОСЛЕДУЮЩЕМ ПРИСВАИВАНИИ ВИДА reference to boolean:

априорное значение получателя = ref-> false не требует приведений
априорное значение источника = true не требует приведений

Состояние стека после исполнения < 0, 1 > := < 0, 0 >:

Display [0] ::
[0] b => true
[1] rb => ref-> true
```

```
Состояние стека после исполнения .ref.ref.bool rrb = .loc.ref.bool:

Display [0] ::
[0] b => true
[1] rb => ref-> true
[2] rrb => ref->ref-> false

В ПОСЛЕДУЮЩЕМ ПРИСВАИВАНИИ ВИДА reference to reference to boolean:

априорное значение получателя = ref->ref-> false не требует приведений
априорное значение источника = ref-> true не требует приведений

Состояние стека после исполнения < 0, 2 > := < 0, 1 >:

Display [0] ::
[0] b => true
[1] rb => ref-> true
[2] rrb => ref->ref-> true

В ПОСЛЕДУЮЩЕМ ПРИСВАИВАНИИ ВИДА reference to boolean:

априорное значение получателя = ref-> true не требует приведений
априорное значение источника = ref->ref-> true
апостериорное значение источника = true

Состояние стека после исполнения < 0, 1 > := < 0, 2 >:

Display [0] ::
[0] b => true
[1] rb => ref-> true
[2] rrb => ref->ref-> true

ТЕКУЩЕЕ ОКРУЖЕНИЕ ПОСЛЕ ВЫХОДА ИЗ БЛОКА ТЕКУЩЕГО УРОВНЯ

ПУСТО

ПРОГРАММА ВЫПОЛНЕНА!
```

Рис. 1. Протокол исполнения программы `eq_dec_test_1`

10. ПРИМЕР 2: РАЗЫМЕНОВАНИЕ ИМЁН И ОБОБЩЕНИЕ ЦЕЛЫХ ДО ВЕЩЕСТВЕННЫХ

История, описываемая следующей программой на Алголе 68, такова.

Создаются целая константа *i* (посредством исполнения изображения целого 5) и вещественная переменная *x* (посредством исполнения генератора вещественного).

Затем переменной *x* присваивается целое *i*, которое предварительно обобщается.

Далее с помощью генератора создаётся косвенное имя вещественного *y*, которому затем присваивается значение *x*. Это присваивание не требует приведений, так как в нём виды получателя и источника удовлетворяют условию (1) параграфа 4 при MODE = *reference to real*.

Далее создаются целая константа *j* (посредством изображения целого 17) и имена целых *jj* и *jjj* видов *reference to integral* и *reference to reference to integral* (посредством генераторов соответствующего вида), затем три присваивания, два из которых не требуют приведений, а последнее – последовательности приведений косвенного имени *jjj*: два раза разыменовать и обобщить.

Динамика событий при построении семантического дерева программы и её исполнения показана в протоколе.

Листинг II. Построение семантического дерева программы
.begin .int i = 5; .ref.real x = .loc.real; x := i;
.ref.ref.real y = .loc.ref.real; y := x;
.int j = 17;.ref.int jj = .loc.int;.ref.ref.int jjj = .loc.ref.int;
jj := j; jjj := jj; x := jjj .end и её исполнения.

```
program eq_dec_test_2;
uses CRT, objects, Strings,
     VALUES, PLAIN_VALUES, STANDART, CONSTRUCTS,
     CLAUSES, ENVIRON,DECLARATIONS, COERCION;
var  ids, id1s, int_default, ref_to_int_default, ref_to_ref_to_int_default,
     real_default, ref_to_real_default, r : string;
     id, id1 : PIntegralDenotation;
     i, j, jj, jjj, x, y : PTag;
     TagList1, TagList2, TagList3, TagList4, TagList5, TagList6 : PTagList;
     rg : PRealGenerator;
     rrg : PReference_to_real_Generator;
     ig: PIntegralGenerator;
     ig1 : PReference_to_integral_Generator;
     IdentityDeclaration1, IdentityDeclaration2,
     IdentityDeclaration3, IdentityDeclaration4,
     IdentityDeclaration5, IdentityDeclaration6 : PIdentityDeclaration;
     Addr00, Addr01, Addr02, Addr03, Addr04, Addr05 : PAddr;
     Assignment, Assignment4 : PRealAssignment;
     Assignment1: PReference_to_Real_Assignment;
     Assignment2 : PIntegralAssignment;
     Assignment3: PReference_to_Integral_Assignment;
     cl0 : PConstructList;
     Range0 : PRange;
     main : PClosedClause;
begin ClrScr;
 writeln (#13#10'Тестирование программы Алгола 68:');
 writeln ('.begin .int i = 5; .ref.real x = .loc.real; x := i;');
 writeln ('.ref.ref.real y = .loc.ref.real; y := x;');
writeln ('.int j=17;.ref.int jj = .loc.int;.ref.ref.int jjj=.loc.ref.int;');
 writeln ('jj := j; jjj := jj; x := jjj .end');
 writeln (#13#10'ПРОСТРАНСТВО ДАННЫХ:'#10#13);
 { СОЗДАНИЕ СТЕКА ДАННЫХ }
   Stack := New (PStack, Init (1));
   writeln ('Стек на ', Stack^.Limit, ' участок');
 { СОЗДАНИЕ ТАБЛИЦЫ DISPLAY }
   Display := New (PDisplay, Init (1));
   writeln ('Display на ', Display^.Limit,' участок'#13#10);
   writeln ('===== СОЗДАНИЕ СЕМАНТИЧЕСКОГО ДЕРЕВА ПРОГРАММЫ ====='#13#10);
 { СОЗДАНИЕ КОНСТРУКЦИЙ БЛОКА 0 }
 { Строки для представления генераторов, изображений целых и присваиваний }
 real_default := '.loc.real';
 ref_to_real_default:= '.loc.ref.real';
 int_default := '.loc.int';
 ref_to_int_default:= '.loc.ref.int';
 ids := '5';
 id1s := '17';
 r := ' := ';
 { Построение конструкций входной программы }
 id := New (PIntegralDenotation, Init (@ids[1], 5));
 writeln ('Изображение целого: ', id^.Show);
```

```
id1 := New (PIntegralDenotation, Init (@id1s[1], 17));
writeln ('Изображение целого: ', id1^.Show);
{ Создание тегов }
writeln ('Создание Tag''a i ');
i := New (PTag, Init (true, 'i', id));
writeln ('Tag: ', i^.Show);
writeln ('Создание Tag''a j ');
j := New (PTag, Init (true, 'j', id1));
writeln ('Tag: ', j^.Show);
rg := New (PRealGenerator, Init (@real_default[1], true));
writeln ('Генератор вещественного: ', rg^.Show);
ig := New (PIntegralGenerator, Init (@int_default[1], true));
writeln ('Генератор имени целого: ', ig^.Show);
writeln ('Создание Tag''a jj ');
jj := New (PTag, Init (true, 'jj', ig));
writeln ('Tag: ', jj^.Show);
ig1 := New (PReference_to_integral_Generator,
            Init (@ref_to_int_default[1], true));
writeln ('Генератор имени имени целого: ', ig1^.Show);
writeln ('Создание Tag''a jjj ');
jjj := New (PTag, Init (true, 'jjj', ig1));
writeln ('Tag: ', jjj^.Show);
writeln ('Создание Tag''a x ');
x := New (PTag, Init (true, 'x', rg));
writeln ('Tag: ', x^.Show);
rrg := New (PReference_to_real_Generator,
            Init (@ref_to_real_default[1], true));
writeln ('Создание Tag''a y ');
y := New (PTag, Init (true, 'y', rrg));
writeln ('Tag: ', y^.Show);
{ Создание списков тегов }
TagList1 := New (PTagList, Init (1, 0));
TagList1^.Insert (i);
writeln (#10#13'Список Tag''ов создан: ', TagList1^.Show);
TagList2 := New (PTagList, Init (1, 0));
TagList2^.Insert (x);
writeln ('Список Tag''ов создан: ', TagList2^.Show);
TagList3 := New (PTagList, Init (1, 0));
TagList3^.Insert (y);
writeln ('Список Tag''ов создан: ', TagList3^.Show);
TagList4 := New (PTagList, Init (1, 0));
TagList4^.Insert (j);
writeln ('   Список Tag''ов создан: ', TagList4^.Show);
TagList5 := New (PTagList, Init (1, 0));
TagList5^.Insert (jj);
writeln ('Список Tag''ов создан: ', TagList5^.Show);
TagList6 := New (PTagList, Init (1, 0));
TagList6^.Insert (jjj);
writeln ('Список Tag''ов создан: ', TagList6^.Show);
{ Создание конструкций 'описание тождества' }
IdentityDeclaration1 := New (PIdentityDeclaration,
                             Init ('.int', TagList1));
write (#13#10'Конструкция описание тождества создана: ');
writeln (IdentityDeclaration1^.Show);
IdentityDeclaration2 := New (PIdentityDeclaration,
                             Init ('.ref.real', TagList2));
write ('Конструкция описание тождества создана: ');
```

```
writeln (IdentityDeclaration2^.Show);
IdentityDeclaration3 := New (PIdentityDeclaration,
                             Init ('.ref.ref.real', TagList3));
write ('Конструкция описание тождества создана: ');
writeln (IdentityDeclaration3^.Show);
IdentityDeclaration4 := New (PIdentityDeclaration,
                             Init ('.int', TagList4));
write ('Конструкция описание тождества создана: ');
writeln (IdentityDeclaration4^.Show);
IdentityDeclaration5 := New (PIdentityDeclaration,
                             Init ('.ref.int', TagList5));
write ('Конструкция описание тождества создана: ');
writeln (IdentityDeclaration5^.Show);
IdentityDeclaration6 := New (PIdentityDeclaration,
                             Init ('.ref.ref.int', TagList6));
write ('Конструкция описание тождества создана: ');
writeln (IdentityDeclaration6^.Show);
{ Создание адресов значений в стеке }
Addr00 := New (PAddr, Init (0, 0)); {i}
writeln 'Создан адрес Addr00: ', Addr00^.Show);
Addr01 := New (PAddr, Init (0, 1)); {x}
writeln ('Создан адрес Addr01: ', Addr01^.Show);
Addr02 := New (PAddr, Init (0, 2)); {y}
writeln ('Создан адрес Addr02: ', Addr02^.Show);
Addr03 := New (PAddr, Init (0, 3)); {j}
writeln ('Создан адрес Addr03: ', Addr03^.Show);
Addr04 := New (PAddr, Init (0, 4)); {jj}
writeln ('Создан адрес Addr04: ', Addr04^.Show);
Addr05 := New (PAddr, Init (0, 5)); {jjj}
writeln ('Создан адрес Addr05: ', Addr05^.Show);
{ Создание конструкции присваивание: x := i }
Assignment := New (PRealAssignment,
                   Init (@r[1], nil, nil, Addr01, Addr00, '', 'd'));
writeln (#10#13'Конструкция присваиавние создана: ', Assignment^.Show);
{ Создание конструкции присваивание: y := x }
Assignment1 := New (PReference_to_Real_Assignment,
                    Init (@r[1], nil, nil, Addr02, Addr01, '', ''));
writeln ('Конструкция присваиавние создана: ', Assignment1^.Show);
{ Создание конструкции присваивание: jj := j }
Assignment2 := New (PIntegralAssignment,
                    Init (@r[1], nil, nil, Addr04, Addr03, '', ''));
writeln ('Конструкция присваиавние создана: ', Assignment2^.Show);
{ Создание конструкции присваивание: jjj := jj }
Assignment3 := New (PReference_to_Integral_Assignment,
                    Init (@r[1], nil, nil, Addr05, Addr04, '', ''));
writeln ('Конструкция присваиавние создана: ', Assignment3^.Show);
{ Создание конструкции присваивание: x := jjj }
Assignment4 := New (PRealAssignment,
                   Init (@r[1], nil, nil, Addr01, Addr05, '', 'aad'));
writeln ('Конструкция присваиавние создана: ', Assignment^.Show);
{ Создание списка конструкций блока 0 }
cl0 := New (PConstructList, Init (11, 0));
cl0^.Insert (IdentityDeclaration1);
cl0^.Insert (IdentityDeclaration2);
cl0^.Insert (Assignment);
cl0^.Insert (IdentityDeclaration3);
cl0^.Insert (Assignment1);
```

```
 cl0^.Insert (IdentityDeclaration4);
 cl0^.Insert (IdentityDeclaration5);
 cl0^.Insert (IdentityDeclaration6);
 cl0^.Insert (Assignment2);
 cl0^.Insert (Assignment3);
 cl0^.Insert (Assignment4);
 Range0 := New (PRange, Init (0, 10, cl0));
 main := New (PClosedClause, Init (Range0));
 writeln ('СЕМАНТИЧЕСКОЕ ДЕРЕВО ПРОГРАММЫ СОЗДАНО:', main^.Show);
 writeln ('ЗАПУСК ПРОГРАММЫ ... ');
 main^.Run;
 writeln ('ПРОГРАММА ВЫПОЛНЕНА!'); readln;
end.
```

```
  Тестирование программы Алгола 68:
  .begin .int i = 5; .ref.real x = .loc.real; x := i;
  .ref.ref.real y = .loc.ref.real; y := x;
  .int j = 17; .ref.int jj = .loc.int; .ref.ref.int jjj = .loc.ref.int;
  jj := j; jjj := jj; x := jjj .end

 ПРОСТРАНСТВО ДАННЫХ:

 Стек на 1 участок
 Display на 1 участок

 ======= СОЗДАНИЕ СЕМАНТИЧЕСКОГО ДЕРЕВА ПРОГРАММЫ =======

 Изображение целого: 5
 Изображение целого: 17
 Создание Tag'а i
 Tag: i = 5
 Создание Tag'а j
 Tag: j = 17
 Генератор вещественного: .loc.real
 Генератор имени целого: .loc.int
 Создание Tag'а jj
 Tag: jj = .loc.int
 Генератор имени имени целого: .loc.ref.int
 Создание Tag'а jjj
 Tag: jjj = .loc.ref.int
 Создание Tag'а x
 Tag: x = .loc.real
 Создание Tag'а y
 Tag: y = .loc.ref.real

 Список Tag'ов создан: i = 5
 Список Tag'ов создан: x = .loc.real
 Список Tag'ов создан: y = .loc.ref.real
 Список Tag'ов создан: j = 17
 Список Tag'ов создан: jj = .loc.int
 Список Tag'ов создан: jjj = .loc.ref.int

 Конструкция описание тождества создана: .int i = 5
 Конструкция описание тождества создана: .ref.real x = .loc.real
 Конструкция описание тождества создана: .ref.ref.real y = .loc.ref.real
 Конструкция описание тождества создана: .int j = 17
 Конструкция описание тождества создана: .ref.int jj = .loc.int
 Конструкция описание тождества создана: .ref.ref.int jjj = .loc.ref.int

 Создан адрес Addr00: < 0, 0 >
 Создан адрес Addr01: < 0, 1 >
 Создан адрес Addr02: < 0, 2 >
 Создан адрес Addr03: < 0, 3 >
 Создан адрес Addr04: < 0, 4 >
 Создан адрес Addr05: < 0, 5 >

 Конструкция присваивние создана: < 0, 1 > := < 0, 0 >
 Конструкция присваивние создана: < 0, 2 > := < 0, 1 >
 Конструкция присваивние создана: < 0, 4 > := < 0, 3 >
 Конструкция присваивние создана: < 0, 5 > := < 0, 4 >
 Конструкция присваивние создана: < 0, 1 > := < 0, 0 >
```

```
СЕМАНТИЧЕСКОЕ ДЕРЕВО ПРОГРАММЫ СОЗДАНО:
BEGIN(0)
  [0] .int i = 5
  [1] .ref.real x = .loc.real
  [2] < 0, 1 > := < 0, 0 >
  [3] .ref.ref.real y = .loc.ref.real
  [4] < 0, 2 > := < 0, 1 >
  [5] .int j = 17
  [6] .ref.int jj = .loc.int
  [7] .ref.ref.int jjj = .loc.ref.int
  [8] < 0, 4 > := < 0, 3 >
  [9] < 0, 5 > := < 0, 4 >
  [10] < 0, 1 > := < 0, 5 >
END(0)

ЗАПУСК ПРОГРАММЫ ...

ConstructList:
  [0] .int i = 5
  [1] .ref.real x = .loc.real
  [2] < 0, 1 > := < 0, 0 >
  [3] .ref.ref.real y = .loc.ref.real
  [4] < 0, 2 > := < 0, 1 >
  [5] .int j = 17
  [6] .ref.int jj = .loc.int
  [7] .ref.ref.int jjj = .loc.ref.int
  [8] < 0, 4 > := < 0, 3 >
  [9] < 0, 5 > := < 0, 4 >
  [10] < 0, 1 > := < 0, 5 >

Состояние стека после исполнения .int i = 5:

Display [0] ::
[0] i => 5

Состояние стека после исполнения .ref.real x = .loc.real:

Display [0] ::
[0] i => 5
[1] x => ref->  0.000000000000000E+000

В ПОСЛЕДУЮЩЕМ ПРИСВАИВАНИИ ВИДА reference to real:

априорное значение получателя = ref->  0.000000000000000E+000 не требует приведений
априорное значение источника = 5
Выполнено widening
UV =  5.000000000000000E+000
апостериорное значение источника =  5.000000000000000E+000

Состояние стека после исполнения < 0, 1 > := < 0, 0 >:

Display [0] ::
[0] i => 5
[1] x => ref->  5.000000000000000E+000

Состояние стека после исполнения .ref.ref.real y = .loc.ref.real:

Display [0] ::
[0] i => 5
[1] x => ref->  5.000000000000000E+000
[2] y => ref->ref->  0.000000000000000E+000

В ПОСЛЕДУЮЩЕМ ПРИСВАИВАНИИ ВИДА reference to reference to real:

априорное значение получателя = ref->ref->  0.000000000000000E+000 не требует приведений
априорное значение источника = ref->  5.000000000000000E+000 не требует приведений

Состояние стека после исполнения < 0, 2 > := < 0, 1 >:

Display [0] ::
[0] i => 5
[1] x => ref->  5.000000000000000E+000
[2] y => ref->ref->  5.000000000000000E+000

Состояние стека после исполнения .int j = 17:

Display [0] ::
[0] i => 5
[1] x => ref->  5.000000000000000E+000
[2] y => ref->ref->  5.000000000000000E+000
[3] j => 17
```

```
Состояние стека после исполнения .ref.int jj = .loc.int:

Display [0] ::
[0] i => 5
[1] x => ref->  5.0000000000000000E+000
[2] y => ref->ref->  5.0000000000000000E+000
[3] j => 17
[4] jj => ref-> 0

Состояние стека после исполнения .ref.ref.int jjj = .loc.ref.int:

Display [0] ::
[0] i => 5
[1] x => ref->  5.0000000000000000E+000
[2] y => ref->ref->  5.0000000000000000E+000
[3] j => 17
[4] jj => ref-> 0
[5] jjj => ref->ref-> 0

В ПОСЛЕДУЮЩЕМ ПРИСВАИВАНИИ ВИДА reference to reference to integral:

априорное значение получателя = ref->ref-> 0 не требует приведений
априорное значение источника = ref-> 17 не требует приведений

Состояние стека после исполнения < 0, 5 > := < 0, 4 >:

Display [0] ::
[0] i => 5
[1] x => ref->  5.0000000000000000E+000
[2] y => ref->ref->  5.0000000000000000E+000
[3] j => 17
[4] jj => ref-> 17
[5] jjj => ref->ref-> 17

В ПОСЛЕДУЮЩЕМ ПРИСВАИВАНИИ ВИДА reference to real:

априорное значение получателя = ref->  5.0000000000000000E+000 не требует приведений
априорное значение источника = ref->ref-> 17
Выполнено dereferencing
Выполнено dereferencing
Выполнено widening
UV =  1.7000000000000000E+001
апостериорное значение источника =  1.7000000000000000E+001

Состояние стека после исполнения < 0, 1 > := < 0, 5 >:

Display [0] ::
[0] i => 5
[1] x => ref->  1.7000000000000000E+001
[2] y => ref->ref->  5.0000000000000000E+000
[3] j => 17
[4] jj => ref-> 17
[5] jjj => ref->ref-> 17

ТЕКУЩЕЕ ОКРУЖЕНИЕ ПОСЛЕ ВЫХОДА ИЗ БЛОКА ТЕКУЩЕГО УРОВНЯ

ПУСТО

ПРОГРАММА ВЫПОЛНЕНА!
```

Рис. 2. Протокол исполнения программы **eq_dec_test_2**

ЗАКЛЮЧЕНИЕ

Понятие приведения используется во многих языках программирования высокого уровня, ориентированных на структуру данных, а не действий, как например в языках функционального программирования.

В модели вычислений, которая положена в основу метода описания языка Алгол 68 [2], это понятие существенно используется, и его реализация в объектах интересна и поучительна сама по себе.

Литература

1. *Мартыненко Б .К.* Учебный исследовательский проект реализации алгоритмических языков // Компьютерные инструменты в образовании, № 5, 2008. С. 3–18.

2. Под ред. *А. ван Вейнгаарден, Б. Майу, Дж. Пек, К. Костер* и др. Пересмотренное сообщение об Алголе 68. М., 1979. 533 с.

3. *Мартыненко Б. К.* Учебный исследовательский проект реализации алгоритмических языков: значения и конструкции // Компьютерные инструменты в образовании, № 1, 2009. С. 10–25.

4. *Мартыненко Б. К.* Учебный исследовательский проект реализации алгоритмических языков: описания и окружения // Компьютерные инструменты в образовании, № 2, 2009. С. 12–29.

5. *Мартыненко Б. К.* Учебный исследовательский проект реализации алгоритмических языков: генераторы и имена // Компьютерные инструменты в образовании, № 3, 2009. С. 3–17.

6. *Michaël Van Canneyt*. Reference guide for Free Pascal. 2002. 188 p.

Abstract

The implementation of coercions based on the object-oriented model of semantics of the programming language Algol 68 as example is discussed.

Мартыненко Борис Константинович, доктор физико-математических наук, профессор кафедры информатики математико-механического факультета СПбГУ,

mbk@ctinet.ru