Разработка скриптов и язык программирования Ada

Киркоров Сергей Иванович

Аннотация

Материал статьи освещает вопросы создания уникальных сценариев (так называемых скриптов) для различных сред их интерпретации и может быть использован при разработки учебного курса "Системное ПО и программирование". На примере языка программирования Ada продемонстрированы возможности создания уникальных сценариев для операционных систем, СУБД, пакетов прикладных программ типа MATLAB и виртуальной машины Java.

Содержание

Введение

В настоящее время именно среда выполнения предопределяет тот инструментарий с помощью которого специалисты реализуют свои идеи в науке, промышленности и обучении, при подготовке кадров в научной и технических областях. Желание абстрагироваться от технических деталей и сосредоточиться специфике задачи привело к бурному росту, в том числе и узко прикладных языков программирования. Характерной чертой, которых является их жесткая привязка к аппаратно-программной платформе (АПП). Таким образом, целесообразно определить аппаратно-программную платформу как связку: аппаратура-ОС-среда выполнения. В технической и экономической сфере наиболее популярные платформы: виртуальная машина (VM) Java, GCC, Microsoft .NET Framework, WEB-вычисления (сервер Apache, MS IIS). Достижения физико-математических наук аккумулируются в таком инструментарии, как Mathematica (фирма WOLFRAMRESEARCH), Mathcad (фирма PTC), Statistica (фирма STATSOFT), Matlab (фирмы MATHWORKS).

Большинство задач выполняемых на АПП имеют графические интерфейсы пользователей (GUI). Однако для того, чтобы раскрыть все возможности АПП необходимо уметь создавать и использовать сценарии – программу для интерпретатора команд АПП. Принято считать интерпретаторы команд бывают двух типов [1-5]:

1. **Простой интерпретатор** анализирует и тут же выполняет (собственно интерпретация) программу покомандно (или построчно), по мере поступления её исходного кода на вход интерпретатора;
2. **Интерпретатор компилирующего типа** — это система из компилятора, переводящего исходный код программы в промежуточное представление, например, в байт-код или p-код, и собственно интерпретатора, который выполняет полученный промежуточный код (так называемая виртуальная машина).

Достоинством простого интерпретатора является мгновенная реакция. Недостаток — такой интерпретатор обнаруживает ошибки в тексте программы только при попытке выполнения команды (или строки) с ошибкой [5].

Достоинством интерпретатора компилирующего типа является большее быстродействие выполнения программ (за счёт выноса анализа исходного кода в отдельный, разовый проход, и минимизации этого анализа в интерпретаторе). Недостатки — большее требование к ресурсам и требование на корректность исходного кода [5].

Некоторые интерпретаторы (например, для Matlab, MySQL и языков Лисп, Scheme, Python, Бейсик и других) могут работать в режиме диалога или так называемого цикла чтения-вычисления-печати (англ. *read-eval-print loop*, *REPL*). В таком режиме интерпретатор считывает законченную конструкцию языка (например, s-expression в языке Лисп), выполняет её, печатает результаты, после чего переходит к ожиданию ввода пользователем следующей конструкции [5].

Не смотря на удобство использования выше описанного инструментария, возникают трудности в создании реальных систем или проведения вычислительного эксперимента для разработанной математической модели. Основная проблема – завышенные требования к производительности ЭВМ, ограничения в объеме обрабатываемых данных, низкая скорость обработки данных, отсутствие механизмом обеспечивающих защиту обрабатываемой информации и т.д. Поэтому применение этого инструментария ограничиваются на начальном этапе разработки. С другой стороны лицензионные ограничения и относительно высокая стоимость инструментария является существенным фактором, который сужает область использования этого инструментария.

Одним из решением задачи оптимизации время/деньги – интеграция подходов и АПП на самом раннем этапе подготовки к проведению вычислительного эксперимента и при создания реальных систем. Своеобразным "программным клеем" целесообразно использовать язык программирования Ada. Реализацией идеи интеграции является расширяемая платформа OEM-2011 for Windows разработанная на языке программирования Ada [6]. Значительно сократить время разработки позволяют сценарии для АПП, которые при необходимости в будущем могут быть реализованы как полноценные исполняемые модули, созданные на языке программирования Ada для той же самой или другой АПП.

Расширяемая платформа OEM-2011 for Windows – совокупность взаимоувязанных пакетов языка Ada. OEM-2011 встраивается в графическую интерактивную систему разработки программ (IDE) - GPS фирмы AdaCore. В результате получаем среду разработки и отладки программ. Отлаженное ПО исполняется на "естественном" программном интерфейсе (API) Win32, может использовать бинарные модули COM и динамические библиотеки DLL, вызывать подсистемы и сам инструментарий например такой как Matlab. В совокупности OEM-2011 – платформа которая позволяет создавать распределенные графические 2D/3D и консольные/серверные приложения в том числе производить "облачные" вычисления (SaaS), взаимодействовать в реальном времени с лабораторным оборудованием или с АСУТП, использовать возможности научного программного инструментария без ущерба производительности системе, создания интерактивных 3D-stereo тренажеров и АРМ-ов дистанционного управления в реальном времени критическими объектами (в отличии от VM Java и Microsoft .NET Framework).

Способность к расширяемости ОЕМ-2011 обеспечивается самим языком программирования Ada и архитектурой взаимосвязанных пакетов.

В то же время, AdaCore GNAT GPL 2009/2010 – инструментальные средства для разработки исполняемых модулей языка Ada-2005 содержат библиотеки и утилиты облегчающие организацию взаимодействия с интерпретатором, в том числе команд операционной системы (ОС) [7,8,9]. Ниже материал работы базируется на переводе статей из серии "Ada Gem of the Week" написанных сотрудниками AdaCore, экспертами по языку программирования Ada-2005 [7-15].

Как организовать поиск текста

В стандарте языка программирования Ada определено несколько подпрограмм, которые должны быть реализованы в библиотеках поставляемых вместе с транслятором, имеющих отношение к поиску текста в строке. В дополнение к ним AdaCore GNAT GPL 2009/2010 поставляет дополнительные пакеты для поиска. Ниже мы рассмотрим некоторые из них.

Мы предполагаем, что будем искать текст внутри строки которая находится в оперативной памяти. В случае необходимости поиска текста в файле, мы просто загружаем его в память для того чтобы выполнить поиск текста и получить результирующую строку.

Мы также предполагаем, что будем использовать для хранения текста тип данных `String` и не будем иметь дело с `Unbounded_String` или `Bounded_String`, так как их легко преобразовать в `String`. Предопределённые подпрограммы в стандартном пакете Ada определены для всех трёх типов, но в спецификации пакета GNAT определены только подпрограммы для типа данных `String`. Конечно, конвертация строк непременно приводит к неэффективности кода, и это приведет к указанию GNAT на спецификации пакетов `Ada.Strings.Unbounded.Aux` которые позволят Вам считывать элементы данных, включенные в `Unbounded_String`, поэтому Вы должны явно применять это преобразование при необходимости.

Однако, текст, который вы используете для поиска (другими словами "шаблон" поиска), может иметь различные форматы. Это может быть фиксированный текст, который должен быть найдет в точности, как есть, или это может быть регулярное выражение, которое вычисляется в процессе создания множества строк поиска.

Когда производится поиск для фиксированного текста, то очевидным кандидатом является предопределенные подпрограммы из пакета `Ada.Strings.Fixed` библиотеки периода выполнения программы. Все значимые подпрограммы начинаются с префикса `Index`. Эти подпрограммы корректно оптимизированы, хотя они не написаны специально на ассемблере для ускорения скорости выполнения. Похожие программы существуют и для `Unbounded_String` и `Bounded_String`.

Другие усовершенствованные алгоритмы существуют для поиска фиксированного текста внутри очень длинного текста. Эти алгоритмы тратят часть времени на исследование во время инициализации шаблона, создавая внутреннее представление (например для создания автомата с конечным числом состояний), для того чтобы потом поиск был более эффективным. Например это алгоритм Boyer-Moore.

Такие предварительные вычисления могут быть более затратные по времени, когда это делается для поиска в случае если строки небольшие, поэтому использование подпрограмм `Index` в этом случае более эффективное решение. Однако как только размер строки увеличивается, использование усовершенствованных алгоритмов становится выгодным. Библиотека времени исполнения GNAT в текущий момент не поставляют реализации таких алгоритмов, но GPS IDE (интерактивная среда разработки и отладки программ GPS) имеют такие пакеты, которые в будущем будут легко интегрированы в библиотеку исполнения GNAT. Сообщите в AdaCore если интересуетесь такими пакетами.

Регулярные выражения, которые являются хорошо известной особенностью некоторых языков программирования, например Perl, и обеспечивают расширенные возможности поиска.

Они поддерживают синтаксис для описания шаблонов, которые должны соответствовать строкам поиска. Например, “a*b” должны быть найдены все строки начинающиеся "a" и заканчивающиеся "b", нулевым или с некоторым количеством символов между ними. Также, “a[bc]d” должны быть найдены все трех символьные строки начинающиеся с "a" и заканчивающиеся "d" с символами "b" или "c" между ними. Мы не будем дальше углубляться в вопрос, как писать регулярные выражения, так как существует в Интернете масса руководств по языку Perl и другим языкам где этот вопрос детально освещён.

GNAT обеспечивает двумя пакетами для поиска с помощью регулярных выражений.

Первый из них который мы будем рассматривать дальше – это `GNAT.Regexp`. Он не является полным, в смысле пакета регулярных выражений, так как он не поддерживает образцы для поиска подстрок соответствующих скобочных (интервальных) групп. Он также пытается проверить весь текст как есть, шаблону “a[bc]d” будет соответствовать только текст если он будет длиной три символа. Пакет поддерживает две версии регулярных выражений: обычная, которая описана выше, и другая которая обычно называется “globbing” шаблон, и используется совместно с параметрами заданными командной строкой интерпретатора Unix. Однако, пакет `GNAT.Regexp` часто используется для поиска связанных имен файлов.

Вот короткий пример как найти все файлы исходных текстов Ada в текущем каталоге. Как вы заметете, регулярное выражение во первых необходимо скомпилировать в внутреннею форму конечного автомата и затем производит очень эффективное соответствие шаблону посредством функции `Match`.

```
with Ada.Directories; use Ada.Directories;
with GNAT.Regexp;     use GNAT.Regexp;

procedure Search_Files (Pattern : String) is
  Search : Search_Type;
  Ent    : Directory_Entry_Type;
  Re     : constant Regexp := Compile (Pattern, Glob => True);

begin
  Start_Search (Search, Directory => ".", Pattern => "");
  while More_Entries (Search) loop
     Get_Next_Entry (Search, Ent);
     if Match (Simply_Name (Ent), Re) then
         ... --  Matched, do whatever
     end if;
  end loop;
  End_Search (Search);
end Search_Files;

Search_Files ("*.adb");
```

Следующий пакет, который поставляется вместе с `GNAT`, это `GNAT.Regpat` .Пакет обеспечивает полную обработку обычных конструкций регулярных выражений, но не со всеми современными возможностями которые есть в языках подобным Perl, можно посмотреть например такие как `look-ahead`, `unicode`,… так далее. Регулярные выражения во первых компилируются в байт код, который будет обеспечивать быстрый поиск соответствия текста шаблону. Заметим что этот пакет менее эффективный по сравнению с `GNAT.Regexp`, так как некоторые шаблоны будут читать символы, исследуя их не выдавая результат соответствия, затем возвращается на начальную позицию и продолжает с некоторой другой позиции. На практике эффективность в целом вполне достаточная.

Одной из особенностью этого пакета, что разработан поверх функций пакета `GNAT.Regexp`. обеспечивая поиск соответствия, известный как скобочное групповое соответствие (parenthesis

groups matched). Например, в регулярном выражении “a (big|small) application“ возможно опознать два альтернативных соответствия, как ниже приведенном примере:

```
with GNAT.Regpat;   use GNAT.Regpat;
with Ada.Text_IO;   use Ada.Text_IO;

procedure Search_Regexp (Pattern : String; Search_In : String) is
  Re      : constant Pattern_Matcher := Compile (Pattern);
  Matches : Match_Array (0 .. 1);

begin
  Match (Re, Search_In, Matches);
  if Matches (0) = No_Match then
     Put_Line ("The pattern did not match");
  else
     Put_Line ("The application really is "
               & Search_In (Matches (1).First .. Matches (1).Last));
  end if;
end Search_Regexp;

Search_Regexp ("a (big|small) application",
               "Example of a small application for searching");
```

Несмотря на то что регулярные выражения являются действительно мощными и как правило более чем достаточны для большинства приложений, тем не менее они имеют ограничения в том что могут охватить только основы свободной грамматики не ограниченного контекста. Классическим примером служит невозможность определить правильного количества открывающих и закрывающихся скобок в строке, как например в “(a(b))”, когда количество скобок заранее не известно.

GNAT поставляется с пакетом GNAT.Spitbol.Patterns , который содержит несколько подпрограмм, которые могут быть использованы для реализации сопоставления (пробы на совместимость) для грамматик со свободой контекста. SPITBOL это собственно и есть полный язык программирования, который GNAT может эмулировать посредством подпрограмм из пакета GNAT.Spitbol , но сейчас мы будем рассматривать только его возможности в части сопоставления с шаблонами.

Приведём маленький пример использования GNAT.Spitbol.Patterns. Задача решаемая в этом примере это проверка задана ли строка начинающаяся с открывающейся скобки "[" или "{" и имеет соответствующую закрывающую скобку.

Текст такого шаблона в BNF формате должен быть следующим:

```
  ELEMENT ::= <any character other than [] or {}>
              | ‘[' BALANCED_STRING ']‘
              | ‘{‘ BALANCED_STRING ‘}’
  BALANCED_STRING ::= ELEMENT {ELEMENT}
```

выполнит трансляцию ниже следующая spitbol программа (чтобы узнатьболее подробно о формате шаблонов необходимо смотреть информацию на пакет GNAT.Spitbol.Patterns). Заметим, для справедливости, что данная программа не просматривает строку до конца, чтобы проверить баланс скобок.

```
with GNAT.Spitbol.Patterns;  use GNAT.Spitbol.Patterns;

procedure Search_Spitbol is
  Element, Balanced_String : aliased Pattern;
  At_Start : Pattern;
```

```
begin
  Element := NotAny ("[]{}")
        or ('[' & (+Balanced_String) & ']')
        or ('{' & (+Balanced_String) & '}');
  Balanced_String := Element & Arbno (Element);
  At_Start := Pos (0) & Balanced_String;
  Match ("[ab{cd}]",   At_Start); --  True
  Match ("[ab{cd}",    At_Start); --  False
  Match ("ab{cd}",     At_Start); --  True
end Search_Spitbol;
```

Возможности работы со скриптами в GNAT

Первое что обычно делает программа – это разбирает свою командную строку, чтобы найти те особенности, которые пользователь хочет активизировать. Стандартом языка Ada определен пакет `Ada.Command_Line`, который по существу дает Вам доступ к каждому из аргументов в командной строке. Но GNAT поставляет значительно более расширенный функционально пакет `GNAT.Command_Line`, который помогает функционально манипулировать этими аргументами, так что вы легко можете выделить ключи, их (возможно опциональные) аргументы и любые оставшиеся аргументы. Этот короткий пример использует этот пакет:

```
with GNAT.Command_Line;  use GNAT.Command_Line;
procedure Main is
begin
  loop
     case Getopt ("h f: d? e g") is
       when 'h' => Display_Help;
       when 'f' => Set_File (Parameter);
       when 'd' => Set_Directory (Parameter);
       when 'e' | 'g' => Do_Something (Full_Switch);
       when others => exit;
     end case;
  end loop;
exception
  when Invalid_Switch | Invalid_Parameter =>
     Display_Help;
end Main;
```

Это приложение принимает в обработку несколько ключей:

- "-f" требующий внешнего аргумента;
- "-d" для которого, в отличии от предыдущего, аргумент может быть, а может и не быть.

Приложение может вызываться несколькими путями. Например: “-ffile -e -g”, “-f file -e -g”, or “-f file -eg”. Пакет GNAT.Command_Line максимально легко приспосабливаемый в части допустимости вида аргументов задаваемых пользователем, это может помочь сделать приложение легким в использовании.

Другим удобным пакетом является `GNAT.Regpat`, который обеспечивает мощную обработку регулярных выражений в процессе выполнения Ada программы. Этот пакет мы рассмотрели в предыдущем разделе.

Часто встречаемым случаем – это когда необходимо провести синтаксический анализ скрипта в текстовом файле. Несмотря на то что стандартный `Ada.Text_IO` пакет даёт доступ к таким файлам, он имеет несколько недостатков. Первый – он довольно медленный. Фактически, стандарт Ada принуждает этот пакет управлять некоторым количеством

дополнительными элементами данных внутренней информации, которые могут добавить значительные накладные расходы. Также файл читает порцию элементарных данных посредством порций данных того же размера. Этот способ чтения в большинстве систем является медленным (требуется серия блочных вызовов ввода/вывода к системе). Где возможно, лучше использовать более эффективные стандартные пакеты, такие как `Stream_IO`. Кроме того `Ada.Text_IO` не поддерживает какие-либо возможности синтаксического анализа. Технология GNAT Reusable Components (изначально реализована под заказ в июле 2008 года) поддерживается пакетом `GNATCOLL.Mmap`. Этот пакет обычно используется для повышения эффективности системных вызовов при чтении файлов, и как результат значительно повышается скорость чтения серий элементарных данных из текстового файла.

```
with GNATCOLL.Mmap;  use GNATCOLL.Mmap;
procedure Main is
   File : Mapped_File := Open_Read ("filename");
   Str  : Str_Access;
begin
   Read (File);  --  read whole file at once
   Str := Data (File);
   --  you are now manipulating an in-memory version of the file
   Close (File);
end Main;
```

Этот пакет также обеспечивает поддержку чтения только части файла за один раз в память, которая является важной, если вы манипулируете гигантскими файлами. Но часто более эффективно просто читать весь файл целиком зараз.

Касательно возможностей синтаксического анализа, другой пакет который оказывает помощь программистам – это `GNAT.AWK`. Этот пакет поддерживает интерфейсный симулятор к стандартной утилите AWK операционной системы UNIX. Конкретно – это её режим являющимся скрытым циклом поверх целого файла. Шаблон поиска соответствия прикладывается к каждой выданной строке, и когда находится соответствие шаблону поиска, то выполняется "обратный вызов процедуры" (a callback action). Главное здесь, что вы не производите синтаксический разбор сами. Несколько типов итераций поддерживаются, Мы рассмотрим только один из них. С остальными примерами можно ознакомиться в интерфейсном файле `g-awk.ads`.

```
with GNAT.AWK;  use GNAT.AWK;
procedure Main is
   procedure Action1 is
   begin
      Put_Line (Field (2));
   end Action2;

   procedure Default_Action is
   begin
      null;
   end Default_Action;
begin
   Register (1, "saturn|earth", Action1'Access);
   Register (1, ".*", Default_Action'Access);
   Parse (";", "filename");
end Main;
```

Каждая строка файла, обозначенного именем “filename”, содержит группы полей, с использованием точки с запятой как разделитель групп. Регистрируется операция для

выделения значения первого поля в группе. Когда первое поле соответствует “saturn” или “earth”, соответствующее действие выполняется, и второе поле печатается. Для всех остальных случаев ничего не делается. Этот код конечно больше чем эквивалентный для интерпретатора команд ОС UNIX использующий утилиту awk, но вполне разумного размера. Конечно если вы сделаете ошибку в ваше программе, очень возможно что компилятор поможет вам найти их (каждое испытание 20-строчной awk программы, не отсылает к более длинным программам, так что где цена становится выше в результате?).

Дальше мы рассмотрим возможности работы с скриптами для манипулирования внешними процессами.

Многие приложения имеют необходимость выполнить команды на машине. При этом существует несколько случаев или другими словами условий выполнения этой команды: программа может захотеть ожидать завершения команды (например, команда генерирует файл, который приложение затем должно обработать); это может быть команда spawn в фоновом режиме и продолжающее своё выполнения в промежутках между событиями; или ей может быть необходимо взаимодействие с внешним приложением (послать данные в входной поток внешней программы и читать её выходной поток).

Породить процесс, который не взаимодействует с программой, легко. Библиотека времени исполнения GNAT включает пакет GNAT.OS_Lib, который содержит некоторое количество подпрограмм, для низко уровневого интерфейса к системе. В частности, пакет снабжает несколькими подпрограммами с именами Spawn и Non_Blocking_Spawn, в соответствии с именами, будут порождать внешнею программу, соответственно с ожиданием и без ожидания завершения этой программы.

```
with GNAT.OS_Lib;  use GNAT.OS_Lib;

procedure Main is
   Command : constant String := "myapp -f 'a string with spaces';";
   --  We assume the slightly more complex case where the arguments of the
   --  command are part of the same string (this is generally the case when
   --  the command is provided interactively by the user).

   Args        : Argument_List_Access;
   Exit_Status : Integer;
   Pid         : Process_Id;
   Success     : Boolean;

begin
   --  Prepare the arguments. Splitting properly takes quotes into account.

   Args := Argument_String_To_List (Command);

   --  Spawn the command and wait for its possible completion

   if Background_Mode then
      Pid := Non_Blocking_Spawn
         (Program_Name => Args (Args'First).all,
          Args         => Args (Args'First + 1 .. Args'Last));

      --  We could also wait for completion:
      --  Wait_Process (Pid, Success);

   else
      Exit_Status := Spawn
         (Program_Name => Args (Args'First).all,
          Args         => Args (Args'First + 1 .. Args'Last));
```

```
   end if;

   --  Free memory

   Free (Args);

end Main;
```

При дальнейшем рассмотрении пакета GNAT.OS_Lib, можно собрать полезные сведения по оставшимся подпрограммам. Они обеспечивают системно-независимые манипуляции с файлами (проверка существует ли файл, директорий это или символическая ссылка, дата/время последнего доступа модификации файла, копирования, удаления или переименования файла и так далее). Одна из интересных подпрограмм это Normalize_Pathname, которая обеспечивает разрешение символических ссылок и преобразовывать оболочку файлового имени в зависимости от того может ли быть получен доступ к уникальному имени или с игнорированием различия строчных и прописных букв. Это обеспечивает удобный способ проверки – являются ли два имени файла одним и тем же объектом или нет (управление "..", символическими ссылками и разрешение имен в зависимости от регистра букв является действительно сложно для правильной и в то же время системно независимой реализации программы, пакет GNAT.OS_Lib оказывает помощь в решении этой задачи).

Из трех случаем взаимодействия порожденного процесса с программой мы вначале рассмотрим более запутанный, где приложению необходимо взаимодействовать с порожденным процессом. Такое взаимодействие обеспечивается пакетом GNAT.Expect. Где вы можете породить процесс и во время его исполнения, посылать что-то этому процессу на вход и читать с его выхода. Взаимодействие между двумя процессами производится через каналы (pipes). Другая реализация, базируется на ttys (псевдо терминалах), планируется быть доступной как составная часть GNAT Reusable Components, и проектируется чтобы обеспечить замкнутую (closer) эмуляцию которая происходит когда вы порождаете процесс из терминала.

Ниже приведен пример использования пакета GNAT.Expect.

```
with GNAT.Expect;  use GNAT.Expect;

procedure Main is
   Command : constant String := "gdb myapp";
                               --  Let's spawn a debugger session.
   Pd      : Process_Descriptor;
   Args    : Argument_List_Access;
   Result  : Expect_Match;

begin
  --  First we spawn the process. Splitting the program name and the
  --  arguments is done as in the previous example, using
  --  Argument_String_To_List. Remember to free the memory at the end.

   Args := Argument_String_To_List (Command);

   Non_Blocking_Spawn
      (Pd,
       Command     => Args (Args'First).all,
       Args        => Args (Args'First + 1 .. Args'Last),
       Buffer_Size => 0);

   --  The debugger is now running. Let's send a command.
```

```
   Send (Pd, "break 10");

  --  Then let's read the output of the debugger.
  --  Here, we are expecting any possible non-empty output (hence the
  --  ".+" regexp). We might in fact be expecting a file name that
  --  gdb uses to confirm a breakpoint. The regexp would be something like:
  --   "file (.*), line (\d+)"

   Expect (Pd, Result, Regexp => ".+", Timeout => 1_000);

   case Result is
      when Expect_Timeout => ...;  --  gdb never replied
      when 1  => ...;              --  the regexp matched
          --  We then have access to all the output of gdb since the
          --  last call to Expect, through the Expect_Out subprogram.
   end case;

  --  When we are done, terminate gdb:

   Close (Pd);
end Main;
```

Этот пример только вкратце касается возможностей пакета `GNAT.Expect`. С другой стороны заметим, что во время работы GPS, ваша интегрированная среда разработки использует этот пакет чтобы взаимодействовать с внешними процессами.

Взаимодействие модулей языка Ada с виртуальной машиной Java

Взаимодействие модулей написанных на языке Ada с виртуальной машиной Java (JVM) является очень сложной проблемой. В отличии от C, C++ или Fortran, языки программирования Ada и Java выполняются на двух разных АПП. Java выполняется на JVM. Модули Ada выполняются непосредственно под управлением ОС (или вообще на "голом железе"). Для данного случая нет никакой возможности напрямую встроить Java функции в естественным образом компилируемый код языка Ada, через прагму (указание компилятору). Существует два решения этой проблемы:

1. компилировать программу на языке Ada напрямую в Java байт код, используя для этого GNAT для JVM;
2. использовать интерфейс Java Native Interface (JNI), позволяющий передавать сообщения между АПП ОС и АПП JVM.

Сначала мы рассмотрим второй случай.

Инструментарий AdaCore для Java Native Interface.

Использование уровня Java Native Interface без дополнительного инструментария – это утомительный процесс и который может породить много ошибок в программе. К счастью AdaCore поставляет инструментарий GNAT-AJIS для автоматизации генерации интерфейса. в этом разделе будут рассматриваться эти средства, которые позволяют создавать смешанные Ada/Java приложения.

Предположим, что мы имеем некий интерфейс со стороны языка Ada (Application Program Interface - API) и мы хотим вызвать его из Java. Этот API может содержать некоторое множество типов и функций, например:

```
package API is

   type R is record
      F1, F2 : Integer;
```

```
   end record;

   procedure Print (Str : String; V : R);

end API;
```

Вашей целью является – создание объекта типа R в Java коде и затем послать этот объект в процедуру Print написанную на языке Ada, наряду с параметром типа `String`.

Генерация binding code

Сейчас нам необходимо сгенерировать Java код, а также, внешний уровень JNI, который будет соединяться с кодом на языке Ada. Это может быть сделано с используя утилиту ada2java, вызвав её следующим образом:

```
ada2java api.ads –b test –o ada –c java –L my_lib
```

Так как утилита генерирует только обвязку к пакету, то это необходимо только чтобы получить доступ к спецификации пакета на языке Ada (в нашем случае это файл `api.ads`).

Кроме того, для достижения цели в организации соглашений имен для Java, мы используя флаг –b, снабжаем именем базового пакета все сгенерированные классы Java. Здесь все сгенерированные классы и пакеты будут потомками Java пакета “test”.

Флаги –o и –c определяют директории где будет сохраняться сгенерированный код для Ada и Java, соответственно. Здесь мы сохраняем Ada код в директории `ada/` , java код в директории `java/` .

В конце, Ada код основной (здесь это пакет `My_Package`) и сгенерированный склеивается кодом необходимый для компиляции в разделяемую библиотеку которая будет загружаться Java-ой. Для упрощения процесса, возможно сгенерировать файл проекта GNAT, для утилиты gprbuild, который будет управлять компиляцией этой библиотеки, в соответствии с ключом –L.

Написание Java кода

Если вы посмотрите java/ директорий то найдете там связку классов, в частности:

```
test.API.R
test.API.API_Package
test.Standard.AdaString
```

Здесь все необходимые классы для вашего приложения: `test.API.R` отображает Ada объект `R`, `test.API.API_Package` содержит все декларации подпрограмм которые появились из API пакета и не могут быть определены как члены других типов, `test.Standard.AdaString` является автоматически генерируемым классом, который отображает стандартный тип `String`.

Обратим внимание, что класс `test.API`.R предоставляет средства доступа и модификации для полей `F1` и `F2`, а также содержит конструктор по умолчанию. Заметит также, что `test.Standard.AdaString` предоставляет конструктор на базе `java.lang.String`.

Сейчас самое время написать главный Java класс.

```
import test.API.R;
import test.API.API_Package;
import test.Standard.AdaString;
```

```
public class My_Main {

public static void main (String [] argv) {
   R v = new R ();
   v.F1 (10);
   v.F2 (15);
  API_Package.Print (new AdaString (“Hello”), v);
}
}
```

И это всё – это работа как если бы вы написали напрямую вызов из Java к Ada.

Компиляция и выполнение кода

Последний шагом является компиляция и выполнение кода. Так как мы уже создали файл GNAT проекта, для кода Ada, то чтобы скомпилировать его, надо только запустить gprbuild вместе с ним.

```
gprbuild –p –P ada/my_lib.gpr
```

Для того чтобы это работало, необходимо в зависимости от АПП произвести адаптацию: или PATH (для Windows) или LD_LIBRARY_PATH (для Linux/Solaris) чтобы включить в список поиска каталог `ada/lib` . Например (для Linux/Solaris):

```
LD_LIBRARY_PATH=`pwd`/ada/lib/:$LD_LIBRARY_PATH
export LD_LIBRARY_PATH
```

Генерируемый java код должен быть в вашем CLASSPATH, также как и главная программа которую вы написали, а файл lib/ajis.jar в вашей инсталляции AJIS. Предположим что вы будете работать под Linux и основа Java имеет такое же размещение как в файле api.ads. Тогда вам необходимо определить CLASSPATH следующим образом (pwd – команда Linux, которая выдаёт рабочий каталог текущего зарегистрированного пользователя):

```
CLASSPATH=`pwd`:`pwd`/java:<your AJIS installation>/lib/ajis.jar:$CLASSPATH
```

Скомпилируем и запустим Java код сейчас:

```
javac My_Main.java
java My_Main
```

Заметим что библиотеки которые мы скомпилировали, будут автоматически загружаться с помощью сгенерированного оберточного кода.

Создание Ada приложений с вызовами к Java, используя GNAT-AJIS

Мы рассмотрели утилиту ada2java, которая отображает Ada спецификацию в Java спецификацию, чтобы поддержать вызовы из Java пакетов Ada. Хотя утилита ada2java не поддерживает создания Ada-овского построителя спецификации Java, это тем не менее возможно, используя её чтобы выполнить из кода Ada код Java. Первой возможностью сделать это – использование обратного вызова подпрограммы (в терминах Ada языка – вызовы ссылки к подпрограмме, или access-to-subprogram calls).

Рассмотрим следующий API:

```
package API is

   type A_Callback is access procedure (Str : String; Val : Integer);
```

```
   procedure Process (Str : String; C : A_Callback);

end API;

package body API is
   procedure Process (Str : String; C : A_Callback) is
   begin
      C.all (Str, 1);
      C.all (Str, 2);
   end Process;
end API;
```

В этом примере `A_Callback` вызывается дважды из процедуры `Process`. Вашей целью здесь будет обеспечение этими вызовами активизацией Java кода.

Генерация binding code

Аналогично тому как это делали раньше, сгенерируем код:

```
ada2java api.ads –b test –o ada –c java –L my_lib
```

Детальное описание параметров вызова можно найти в документации на утилиту ada2java.

Написание Java кода

Просматривая сгенерированный код, мы увидим, что функция `Process` была определена как следующая:

```
public final class API_Package {
  […]
  static public void Process (test.Standard.AdaString Str, test.API.A_Callback C){
  […]
}

with test.API.A_Callback being:


public abstract class A_Callback {
  […]
  abstract public void A_Callback_Body (test.Standard.AdaString Str, int Val);
  […]
}
```

Заметим, что всё это необходимо для того чтобы реализовать в Java ссылочную подпрограмму на языке Ada. Теперь, путем извлечения описания на `A_Callback`, создан конкретный класс, с реализацией в нем метода `A_Callback_Body` и затем производящий запрос этого объекта к функции `Process`.

Например:

```
import test.API.A_Callback;
import test.API.API_Package;
import test.Standard.AdaString;

public class My_Main {
   static class My_Callback extends A_Callback {
      public void A_Callback_Body (AdaString Str, int Val) {
```

```
         System.out.println (“Call “ + Val + “ of “ + Str);
      }
   }
   public static void main (String [] argv) {
      API_Package.Process (new AdaString (“Hello”), new My_Callback ());
   }
}
```

И в самом деле это легко. Вы устанавливаете процедуру обратного вызова, вместе с Java вызывающей Ada и затем производится обратный вызов к Java.

Компиляция и выполнение кода

Как и раньше, мы сгенерировали файл проекта для сборки, с помощью ключа – L. Для Linux сделаем следующие шаги:

```
gprbuild –p –P ada/my_lib.gpr
LD_LIBRARY_PATH=`pwd`/ada/lib/:$LD_LIBRARY_PATH
export LD_LIBRARY_PATH
CLASSPATH=`pwd`:`pwd`/java:<your AJIS installation>/lib/ajis.jar:$CLASSPATH
javac My_Main.java
java My_Main
```

Если сейчас запустить программу то она не будет полностью работоспособной, как ожидалось. Это связано с тем что генерируется исключение в точке вызова Process на стороне Java. Разберемся в этом более детально.

Решение возникшей проблемы

Проблема которая возникла у нас заключается в том, что нет возможности сохранить объект класса A_Callback, который был создан из под Java – это значит что не существует доступа библиотечного доступа к значениям подпрограмм, раньше чем к объектам подпрограмм. Мы можем посмотреть более детально код который сгенерирован в Ada, но важно здесь то что объекты не могут быть использованы в естественном коде после вызовов к Process, так как они не сохраняются. Другими словами они не могут быть не потеряны.

К сожалению, мы не имеем никакой возможности убедиться в том что объекты действительно ещё доступны, так как AJIS во время выполнения справедливо полагает что ссылается к таким объектам, последовательность которая предохраняет выгрузку объектов проводящую к ошибке полностью лежит на разработчике программы. В этом случае мы должны сделать две вещи. Нет никаких объектов запрашивающие память и вызов к Process является полностью сохраненным. Мы можем дать эту информацию генератору построения посредством AJIS annotations:

```
with AJIS.Annotations;

package API is
   type A_Callback is access procedure (Str : String; Val : Integer);

   procedure Process (Str : String; C : A_Callback);
   pragma Annotate (AJIS, Assume_Escaped, False, Process, “C”);
```

Указание (pragma) означает, что "" параметр в процедуре Process не должен быть назначен как потенциально теряемый, другими словами, разработчик Ada считает под свою ответственность что нельзя позволить ему исчезнуть. AJIS во время исполнения будет делать возможным послать не удаляемые объекты как параметр процедуре.

Заметим что мы сейчас имеем ссылку к исходникам библиотеки AJIS. Простым способом учесть эту ссылку – это использовать её файл проекта:

```
with “ajis”;
project API is
end API;
```

Теперь необходимо пересобрать проект:

```
ada2java api.ads –b test –o ada –c java –L my_lib –P api.gpr
```

После перекомпиляции модулей Ada и Java, мы перезапустим Java и вызов из Ada методов Java будет работать.

Ada/Java перекрестная диспетчеризация

Подробно эта тема раскрыта в "Gem #57: Ada / Java cross dispatching" [12].

Ada/Java обработка исключений

Подробно эта тема раскрыта в "Gem #58: Ada / Java exception handling" [13].

Интерфейс для кода на языке Ada с скриптовым языком Python

Подробно эта тема раскрыта в "Gem #105: Lady Ada Kisses Python — Part 1" и "Gem #106: Lady Ada Kisses Python — Part 2" [14,15].

Заключение

Литература

1. ГОСТ 19781-83; СТ ИСО 2382/7-77 // Вычислительная техника. Терминология: Справочное пособие. Выпуск 1 / Рецензент канд. техн. наук Ю. П. Селиванов. — М.: Издательство стандартов, 1989. — 168 с. — 55 000 экз. — ISBN 5-7050-0155-X
2. *Першиков В. И., Савинков В. М.* Толковый словарь по информатике / Рецензенты: канд. физ.-мат. наук А. С. Марков и д-р физ.-мат. наук И. В. Поттосин. — М.: Финансы и статистика, 1991. — 543 с. — 50 000 экз. — ISBN 5-279-00367-0
3. *Борковский А. Б.* Англо-русский словарь по программированию и информатике (с толкованиями). — М.: Русский язык, 1990. — 335 с. — 50 050 (доп.) экз. — ISBN 5-200-01169-3
4. Толковый словарь по вычислительным системам = Dictionary of Computing / Под ред. В. Иллингуорта и др.: Пер. с англ. А. К. Белоцкого и др.; Под ред. Е. К. Масловского. — М.: Машиностроение, 1990. — 560 с. — 70 000 (доп.) экз. — ISBN 5-217-00617-X (СССР), ISBN 0-19-853913-4 (Великобритания)
5. Википедия, Интерпретатор. Типы интерпретаторов. Электронный ресурс: http://ru.wikipedia.org
6. *С.И. Киркоров*. Новая библиотека OEM как средство обучения и база для доверенных платформ программирования на языке Ada в Win32. Харьков, май 2010 года тезисы доклада на конференции «КОМПЬЮТЕРНОЕ МОДЕЛИРОВАНИЕ В НАУКОЕМКИХ ТЕХНОЛОГИЯХ» (КМНТ-2010) .
7. *Emmanuel Briot, Senior Software Engineer, AdaCore*. Gem #25: How to Search Text. Электронный ресурс: http://www.adacore.com/category/developers-center/gems/ .
8. *Emmanuel Briot, Senior Software Engineer, AdaCore*. Gem #52: Scripting Capabilities in GNAT (Part 1). Электронный ресурс: http://www.adacore.com/category/developers-center/gems/ .

9. *Emmanuel Briot, Senior Software Engineer, AdaCore*. Gem #54: Scripting Capabilities in GNAT (Part 2). Электронный ресурс: http://www.adacore.com/category/developers-center/gems/ .
10. *Quentin Ochem, AdaCore*. Gem #55: Introduction to Ada / Java Interfacing. Электронный ресурс: http://www.adacore.com/category/developers-center/gems/ .
11. *Quentin Ochem, AdaCore*. Gem #56: Creating Ada to Java calls using GNAT-AJIS. Электронный ресурс: http://www.adacore.com/category/developers-center/gems/ .
12. *Quentin Ochem, AdaCore*. Gem #57: Ada / Java cross dispatching. Электронный ресурс: http://www.adacore.com/category/developers-center/gems/ .
13. *Quentin Ochem, AdaCore*. Gem #58: Ada / Java exception handling. Электронный ресурс: http://www.adacore.com/category/developers-center/gems/ .
14. *Emmanuel Briot, AdaCore*. Gem #105: Lady Ada Kisses Python — Part 1. Электронный ресурс: http://www.adacore.com/category/developers-center/gems/ .
15. *Emmanuel Briot, AdaCore*. Gem #106: Lady Ada Kisses Python — Part 2. Электронный ресурс: http://www.adacore.com/category/developers-center/gems/ .
16.