Tipos Abstractos de Datos (TAD)

Programación de Sistemas de Telecomunicación
Informática II

Departamento de Sistemas Telemáticos y Computación (GSyC)

Octubre de 2016

Contenidos

1. Introducción
2. Tipos Abstractos de Datos
3. Definición de TADs mediante Tipos privados y privados limitados
4. Clases de subprogramas de un TAD

Contenidos

1. Introducción
2. Tipos Abstractos de Datos
3. Definición de TADs mediante Tipos privados y privados limitados
4. Clases de subprogramas de un TAD

Tipos en Ada (1)

Tipos escalares

A veces denominados tipos primitivos o tipos básicos.
Los valores u objetos del tipo tienen un único componente que no puede ser descompuesto por el programa en sus partes:

- Tipos numéricos: enteros, de coma flotante, de coma fija
- Tipos enumerados: Character, Boolean
- Tipos acceso (punteros)

- Los tipos escalares numéricos y enumerados tienen los atributos `First`, `Last`, `Range`
- Los tipos escalares discretos (enteros y enumerados) tienen además los atributos `Pos`, `Val`, `Pred`, `Succ`
- En el paquete `Standard` se definen los tipos escalares predefinidos del lenguaje Ada (ver manual de referencia). Este paquete no requiere que se haga `with` de él.

Tipos en Ada (2)

Tipos compuestos

A diferencia de los escalares, los valores de los tipos compuestos están formados por componentes a los que se puede acceder:

- Record: sus componentes son los campos del registro
- Array: sus componentes son los elementos del array

Operaciones de los tipos en Ada

Hay tres operaciones básicas que están predefinidas automáticamente para cualquier tipo en Ada (escalares, compuestos, definidos por el usuario):

- Asignación o copia `:=`
- Comparación de igualdad `=`
- Comparación de desigualdad `/=`

La asignación copia todo el valor, incluyendo sus componentes si es un tipo compuesto (array o record).
Si el valor es un puntero (tipo access) se copia la dirección, no el objeto apuntado.
Las dos comparaciones comparan todo el valor, incluyendo los componentes si es un tipo compuesto (array o record).

Contenidos

1 Introducción

2 Tipos Abstractos de Datos

3 Definición de TADs mediante Tipos privados y privados limitados

4 Clases de subprogramas de un TAD

Tipo Abstracto de Datos (TAD)

¿Qué es un TAD?

Una especificación abstracta de un tipo de datos en la que se describen:

- los valores del tipo
- su interfaz: las operaciones que pueden realizarse con los valores del tipo

Ejemplo: el tipo `Integer`

- Valores:
  `type Mi_Tipo_Enteros is range Min_Integer..Max_Integer`:
  - el rango depende de los bits que usemos para representar cada entero
- Interfaz (operaciones):
  - Aritméticas: `+, -, *, /, rem, mod, **`
  - Comparación: `<, <=, =, /=, >, >=`

Programas cliente que usan TADs

Programa cliente

Los programas que usan un TAD se denominan programas cliente del TAD

1. El código del cliente no debe depender del código fuente de la implementación del TAD, sólo de su interfaz:
   - el cliente no conoce la representación interna de los valores del TAD
   - el cliente no conoce cómo están implementadas las operaciones del TAD
2. El código del TAD no debe depender del código fuente de ningún cliente en particular

Programas cliente que usan TADs

Encapsulación: ventajas de la independencia entre el TAD y los programas cliente

- Si el programador del TAD decide cambiar la implementación de una de las operaciones del TAD el código fuente de los clientes no se ve afectado: no hay que cambiarlo, aunque sí recompilarlo
- Si el programador del TAD decide cambiar la forma en que se representan los valores del tipo tampoco se ve afectado el código fuente de los clientes: no hay que cambiarlo, aunque sí recompilarlo
- El TAD es reusable para múltiples clientes ya que su código no depende de ningún cliente en particular

Representación de los valores del tipo

- Un valor de tipo `Integer` ocupa normalmente lo mismo que el tamaño de palabra del procesador.
- `Variable'Size` es el tamaño en bits de `Variable`.
- Cuando los valores del tipo están compuestos de varios componentes se utilizan tipos compuestos como arrays y registros para representarlos
- Los valores de tipos primitivos como `Integer` se almacenan en variables que deben haberse declarado de tipo `Integer`
- Los valores de un TAD `T` definido por el programador se almacenan en variables que deben haberse declarado de tipo `T`
- A veces utilizamos la palabra objeto para referirnos al valor de un TAD

Mecanismos de Ada para definir TADs

- Subtipos: `subtype Small is Integer range -10..10;`
- Tipos derivados:
  `type Small is new Integer range -100..100;`
- Nuevos tipos enumerados, registros y arrays: Ej:
  `type Message_Type is (Init, Writer, Server);`
- Paquetes: proporcionan encapsulación, ocultación de información
- Tipos private y limited private: refuerzan la encapsulación de los paquetes para que el cliente sólo pueda manipular el objeto a través de la interfaz definida por el TAD
- Sobrecarga de operadores: permite redefinir los operadores aritméticos y de comparación para nuevos tipos
- Excepciones definidas por el usuario: el programador del TAD puede definir excepciones que eleva la implementación del TAD para que el cliente las pueda capturar
- Subprogramas y paquetes genéricos
- Tipos etiquetados: programación Orientada a Objetos (OO) con herencia, polimorfismo y despacho dinámico en Ada

TADs en lenguajes Orientados a Objetos (OO)

- Los conceptos básicos asociados a la programación OO son la encapsulación, el polimorfismo, el despacho dinámico y la herencia.
- En lenguajes orientados a objetos como Java o C++ los TADs se implementan como clases: un concepto que engloba la definición de un nuevo tipo y la encapsulación que proporcionan en Ada los paquetes.
- En C++ o en Java, cada clase define un tipo
- La encapsulación que proporciona la clase la proporciona en Ada el paquete, dentro del que se define uno o más tipos
- En Ada la encapsulación del paquete se refuerza mediante los tipos privados
- En Ada los tipos etiquetados (*tagged*) aportan la herencia, el polimorfismo y el despacho dinámico.

Contenidos

1 Introducción

2 Tipos Abstractos de Datos

3 Definición de TADs mediante Tipos privados y privados limitados

4 Clases de subprogramas de un TAD

Tipos privados en Ada (1)

Tipos privados

En la especificación de un paquete se puede declarar un tipo como private.
La declaración completa del tipo se hace al final del paquete, en la sección private del paquete.

```
package P is
   -- Representación del TAD NO visible para clientes del paquete
   type T is private;
   -- Interfaz definida para el TAD
   procedure P (e: T);
   function  F (e1: T; e2: T) return T;
   ...

private
   -- Representación del TAD visible en el cuerpo del paquete
   type T is new Integer range 1..10;
end P;
```

Tipos privados en Ada (2)

- La definición completa del tipo NO puede ser utilizada por los clientes del paquete en el que se define el tipo privado.
  - el ámbito de las definiciones que aparecen en la parte pública (es decir, antes de la parte privada) es el resto de la parte pública de la especificación, la parte privada, el body, y cualquier paquete o programa que haga with de éste
  - el ámbito de las definiciones de la parte *private* es el resto de la parte *private* y el *body*.
- Los tipos privados tienen sólo 3 operaciones predefinidas: asignación/copia `:=` y los comparadores de igualdad `=` y desigualdad `/=`
  - Ejemplo: Si es un entero, el cliente no puede usar los operadores aritméticos
  - Ejemplo: Si es un registro/array, el cliente no puede acceder a sus componentes/elementos, salvo que algún subprograma de la interfaz del TAD lo permita
- ¿Qué más operaciones se puede realizar con los objetos del tipo privado? ¡Sólo las que se definan en la interfaz!

Ejemplo: Definición de un TAD sin private (1)

```
with Ada.Strings.Unbounded;

package Lists is

   package ASU renames Ada.Strings.Unbounded;

   type Cell;
   type Cell_A is access Cell;

   type Cell is record
      Name : ASU.Unbounded_String;
      Count: Natural := 0;
      Next : Cell_A;
   end record;


   -- Interfaz
   procedure Add (List   : in out Cell_A;
                  A_Name : in     ASU.Unbounded_String;
                  A_Count: in Natural);

   procedure Print_All (List: in Cell_A);

end Lists;
```

Ejemplo: Definición de un TAD sin private (2)

```
with Ada.Text_IO;

package body Lists is
   procedure Add (List   : in out Cell_A;
                  A_Name : in     ASU.Unbounded_String;
                  A_Count: in Natural) is
      P_Aux : Cell_A;
   begin
      P_Aux := new Cell;
      P_Aux.all.Name  := A_Name;
      P_Aux.all.Count := A_Count;
      P_Aux.all.Next  := List;
      List := P_Aux;
   end Add;

   procedure Print_All (List: in Cell_A) is
      P_Aux : Cell_A;
   begin
      P_Aux := List;
      while P_Aux /= null loop
         Ada.Text_IO.Put_Line (ASU.To_String(P_Aux.all.Name) &
                               ": " &
                               P_Aux.all.Count'Img);
         P_Aux := P_Aux.all.Next;
      end loop;
   end Print_All;
end Lists;
```

Ejemplo: Definición de un TAD sin private (3)

```
with Ada.Text_IO;
with Ada.Strings.Unbounded;

with Lists;

procedure Cliente is
  package ASU renames Ada.Strings.Unbounded;

  Mi_Lista: Lists.Cell_A;
begin

  Lists.Add (Mi_Lista, ASU.To_Unbounded_String("Pepe"), 39);
  Lists.Add (Mi_Lista, ASU.To_Unbounded_String("Juan"), 22);
  Lists.Add (Mi_Lista, ASU.To_Unbounded_String("Lola"), 19);

  Ada.Text_IO.Put_Line ("Contenidos de la lista:");
  Lists.Print_All (Mi_Lista);

  Mi_Lista.Next := null; -- ¡ El programador del cliente
                         --   puede escribir esta línea, que compila,
                         --   lo que destroza la implementación de la lista!

  Ada.Text_IO.New_Line;

  Ada.Text_IO.Put_Line ("Contenidos de la lista:");
  Lists.Print_All (Mi_Lista);

end Cliente;
```

Ejemplo: Definición de un TAD sin private (4)

```
$ gnatmake cliente.adb
gcc -c cliente.adb
gcc -c lists.adb
gnatbind -x cliente.ali
gnatlink cliente.ali
$ ./cliente
Contenidos de la lista:
Lola:  19
Juan:  22
Pepe:  39

Contenidos de la lista:
Lola:  19
$
```

Ejemplo: Definición de un TAD con private (1)

```
with Ada.Strings.Unbounded;

package Lists is

   package ASU renames Ada.Strings.Unbounded;

   type Cell_A is private;

   -- Interfaz
   procedure Add (List   : in out Cell_A;
                  A_Name : in     ASU.Unbounded_String;
                  A_Count: in Natural);

   procedure Print_All (List: in Cell_A);

private
   type Cell;
   type Cell_A is access Cell;

   type Cell is record
      Name : ASU.Unbounded_String;
      Count: Natural := 0;
      Next : Cell_A;
   end record;
end Lists;
```

Ejemplo: Definición de un TAD con private (2)

```
$ gnatmake cliente.adb
gcc -c cliente.adb
cliente.adb:19:03: invalid prefix in selected component "Mi_Lista"
gnatmake: "cliente.adb" compilation error
```

- Al haber declarado como privado el tipo `Cell_A`, el cliente no puede usar variables del tipo `Cell_A` como punteros, ¡aunque en realidad lo son!.
  - De hecho, podríamos cambiar el nombre del TAD para que refleje cómo se pretende usar ese tipo y no cómo está implementado:
    `type List_Type is private;`
    mejor que
    `type Cell_A is private;`
- Ahora los clientes sólo pueden hacer lo siguiente con un objeto de tipo Cell_A:
  - Asignarlo `:=`
  - Compararlo `=`, `/=`
  - Lo que nos deje la interfaz (`Add`, `Print_All`)

Tipos limited private

Tipos limited private

En ocasiones ni siquiera queremos que esté definida la asignación ni las comparaciones de igualdad y desigualdad.
Para ello se define el tipo como limited private en lugar de definirlo como private.

Ahora sólo podemos hacer lo siguiente con un objeto de tipo Cell_A:

- Lo que nos deje la interfaz (`Add, Print_All`)
- Nos viene bien que no se pueda copiar una lista mediante la asignación, pues eso provocaría una copia del puntero que apunta a la primera celda, y no una copia de los elementos de la lista.
- Lo mismo para la comparación si lo que queremos es que la comparación se haga entre los elementos de dos listas y no entre los valores de los punteros que apuntan a la primera celda.

Ejemplo: Definición de un TAD con limited private (1)

```
with Ada.Strings.Unbounded;

package Lists is

   package ASU renames Ada.Strings.Unbounded;

   type Cell_A is limited private;

   -- Interfaz
   procedure Add (List   : in out Cell_A;
                  A_Name : in     ASU.Unbounded_String;
                  A_Count: in Natural);

   procedure Print_All (List: in Cell_A);

private
   type Cell;
   type Cell_A is access Cell;

   type Cell is record
      Name : ASU.Unbounded_String;
      Count: Natural := 0;
      Next : Cell_A;
   end record;
end Lists;
```

Ejemplo: Definición de un TAD con limited private (2)

Para hacer copias tenemos que añadir el procedimiento `Lists.Copy`:

```
-- Devuelve una copia de todos los elementos de la lista List en Copy_List
procedure Copy (List: in Cell_A; Copy_List: out Cell_A) is
   P_Aux : Cell_A;
begin
   P_Aux := List;
   while P_Aux /= null loop
      Add (Copy_List, P_Aux.all.Name, P_Aux.all.Count);
      P_Aux := P_Aux.all.Next;
   end loop;
end Copy;
```

En un programa cliente:

```
with Ada.Text_IO;
with Ada.Strings.Unbounded;
with Lists;
procedure Cliente is
  package ASU renames Ada.Strings.Unbounded;
  Mi_Lista, Otra_Lista: Lists.Cell_A;
begin
  Lists.Add (Mi_Lista, ASU.To_Unbounded_String("Pepe"), 39);

  -- La siguiente sentencia no compila por ser limited private
  Otra_Lista := Mi_Lista;

  -- Con copy sí:
  Lists.Copy (Mi_Lista, Otra_Lista);
end Cliente;
```

Contenidos

1 Introducción

2 Tipos Abstractos de Datos

3 Definición de TADs mediante Tipos privados y privados limitados

4 Clases de subprogramas de un TAD

Clases de subprogramas de un TAD

- Constructores de objetos: subprogramas que componen un objeto del TAD a partir de componentes que son pasados como parámetros al constructor.
  - Ejemplo: si la representación del TAD es mediante un registro, el constructor puede ser una función que devuelve un registro relleno a partir de los parámetros del constructor.
- Selectores de componentes: Devuelven un componente particular de los que forman parte de un valor del tipo
- Consultas de los valores: subprogramas que comprueban si un objeto tiene una propiedad
  - Ejemplo: ¿Está vacío?, ¿Está lleno?, etc.
- Entrada/Salida: Comunicación con el exterior de los objetos del tipo
  - Envío/recepción a través de la red o a un fichero de los objetos del tipo, representación en modo texto del objeto para mostrarlo al usuario, etc.

Definición de operadores infijos de un TAD

Pueden definirse operadores infijos para un tipo (sea o no private, o limited private): `+`, `-`, `*`, `/`, `**`, `<`, `<=`, `>`, `>=`

```
package Lists is
   ...
   function "+" (L1: Cell_A; L2: Cell_A) return Cell_A;
```

Desde un programa cliente:

```
with Lists;
procedure Cliente is
   use type Lists.Cell_A;
   L1, L2, L3, L4: Cell_A;
begin
   ...
   L3 := Lists."+" (L1, L2);

   L4 := L1 + L2;    -- Requiere use type Lists.Cell_A;

end Cliente;
```

Ejemplo: TAD Ada.Calendar.Time (1)

```
package Ada.Calendar is

   -- Declaración del tipo privado
   type Time is private;

   -- Definición de subtipos auxiliares
   subtype Year_Number  is Integer range 1901 .. 2099;
   subtype Month_Number is Integer range 1 .. 12;
   subtype Day_Number   is Integer range 1 .. 31;
   subtype Day_Duration is Duration range 0.0 .. 86_400.0;

   -- Constructor
   function Time_Of
     (Year    : Year_Number;
      Month   : Month_Number;
      Day     : Day_Number;
      Seconds : Day_Duration := 0.0)
      return    Time;

   -- Selectores
   function Year    (Date : Time) return Year_Number;
   function Month   (Date : Time) return Month_Number;
   function Day     (Date : Time) return Day_Number;
   function Seconds (Date : Time) return Day_Duration;
   procedure Split
     (Date    : Time;
      Year    : out Year_Number;
      Month   : out Month_Number;
      Day     : out Day_Number;
      Seconds : out Day_Duration);
...
```

Ejemplo: TAD Ada.Calendar.Time (2)

```
...


   -- Consulta
   function Clock return Time;

   -- Operaciones Aritméticas
   function "+" (Left : Time;     Right : Duration) return Time;
   function "+" (Left : Duration; Right : Time)     return Time;
   function "-" (Left : Time;     Right : Duration) return Time;
   function "-" (Left : Time;     Right : Time)     return Duration;

   -- Operaciones de Comparación
   function "<"  (Left, Right : Time) return Boolean;
   function "<=" (Left, Right : Time) return Boolean;
   function ">"  (Left, Right : Time) return Boolean;
   function ">=" (Left, Right : Time) return Boolean;

   -- Excepciones
   Time_Error : exception;

private
   ...

      -- Definición del tipo
   type Time is new Duration;
end Ada.Calendar;
```