As linguagens Ada e Spark

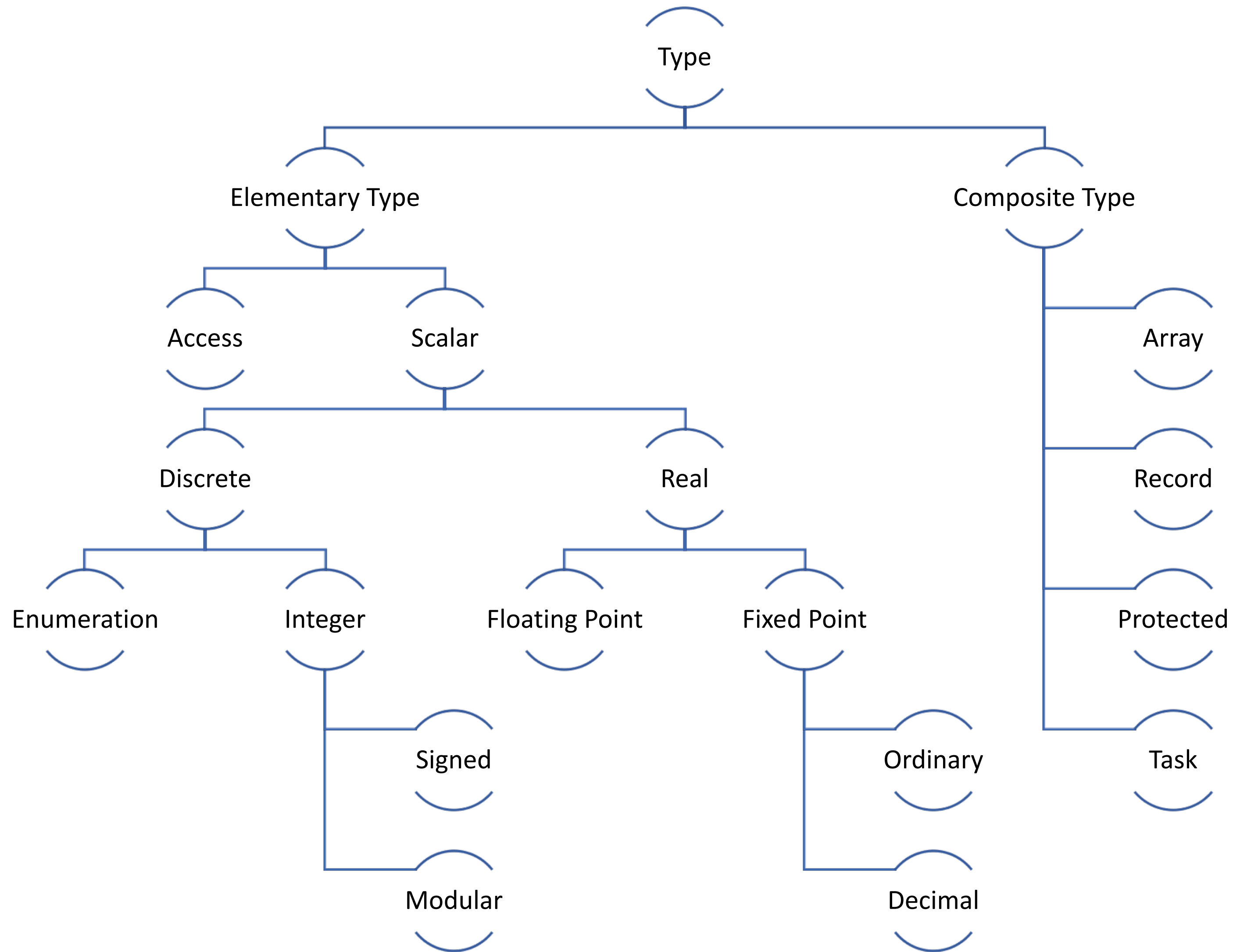
Conceitos de alto-nível

Parte I – Conceitos Introdutórios de Ada 2012

Programa Ada simples

```
-- Imported library
with Ada.Text_IO;

procedure Hello is
   -- Variable declaration
   A, B, C : Integer;
begin
   -- Variable assignments
   A := Integer'Value (Ada.Text_IO.Get_Line);
   B := Integer'Value (Ada.Text_IO.Get_Line);
   --  Addition of the assigned variables
   C := A + B;

   -- Outputs according to the positioning of the
   -- value of variable 'C' in the domain of supported integers
   if C = 0 then
      Ada.Text_IO.Put_Line ("RESULT IS 0");
   elsif C > 0 then
      Ada.Text_IO.Put_Line ("POSITIVE RESULT :" & Integer'Image (C));
   else
      Ada.Text_IO.Put_Line ("NEGATIVE RESULT :" & Integer'Image (C));
   end if;
end Hello;
```

Modelo de tipos do Ada



Tipos simples – discretos e reais

- Como base, temos os tipos elementares, podendo estes ser discretos ou reais (dentro da gama dos “scalar types”):
  - Inteiros (com sinal ou em aritmética módulo)
  - Enumerações
  - Floating point
  - Fixed-point
- Podem ser atribuídas restrições aos tipos simples de forma a garantir maior integridade ao código:

```
type Score is range 0 .. 20;
type Percentage is range -100 .. 100;
```

```
type Color is (Red, Green, Blue, Yellow, Black);
type Ternary is (True, False, Unknown);
```

Tipos simples – declarações

- Podem também ser declarados novos tipos, com base em tipos já existentes, aos quais podem também ser adicionadas restrições:

```
type Math_Score is new Score;
```

```
type Math_Score is new Score range 0 .. 10;
type Primary_Color is new Color range Red .. Blue;
```

- Em alguns casos, pode ser estabelecida uma conversão direta de um valor de um tipo para um valor correspondente noutro tipo, tendo estas conversões que ser explícitas:

```
V1 : Float := 0.0;
V2 : Integer := Integer (V1);
```

```
type T1 is range 0 .. 10;
type T2 is range 1 .. 10;

V1 : T1 := 0;
V2 : T2 := T2 (V1); -- Run-time error!
```

Tipos simples – operações e atributos

- Para os tipos numéricos, estão definidas as operações standard e algumas extra

=	/=	<	<=	>	>=	+	-	*	/	abs	**	mod	rem

- Os tipos têm atributos associados, que podem ser consultados em tempo de execução para dar maiores garantias ao código:

Attribute Name	Documentation
First	Returns the first value of the type
Last	Returns the last value of the type
Image (X)	Converts a value to its corresponding String
Value (X)	Converts a String to its corresponding value
Min (X, Y)	Returns the maximum of two values
Max (X, Y)	Returns the minimum of two values
Pred (X)	Returns the previous value
Succ (X)	Returns the next value
Range	Equivalent of T'First ..T'Last

Attribute Name	Documentation
Pos (X)	Returns the position of the value in the type
Val (X)	Returns a value according to its position

```
S : String := Integer'Image (42);
```

```
V : Character := Character'Val (0);
W : Next_Character := Character'Val (Character'Pos (V) + 1);
```

Instruções e controlo – declarações de variáveis

- A declaração de variáveis em Ada

```
A : Integer;
B : Integer := 5;
C : constant Integer := 78;
D, E : Integer := F (5);
```

- Elaboração das declarações é feita de forma sequencial

```
A : Integer := 5;
B : Integer := A;
C : Integer := D; -- Erro na elaboracao e compilacao
D : Integer := 0;
```

- Inicialização de valores de variáveis (feita de forma individual)

```
A, B : Float := Compute_New_Random;
--  This is equivalent to:
A : Float := Compute_New_Random;
B : Float := Compute_New_Random;
```

- Blocos de declaração (usados quando é conveniente introduzir novas declarações localmente)

```
declare
   A : Integer;
begin
   A := 0;
end;
```

```
declare
   A : Integer;
begin
   null;
end;
```

Instruções e controlo - condicionais

- Blocos if-then-else

```
if A = 0 then
   Put_Line ("A is 0");
elsif B = 0 then
   Put_Line ("B is 0");
else
   Put_Line ("Else... ");
end if;
```

- Expressões condicionais

```
Put_Line ((if A = 0 then "A is 0” elsif B = 0 then "B is 0” else "Else"));
```

- Operadores para formação de asserções lógicas em blocos/expressões condicionais:

=	/=	<	<=	>	>=

- Podemos combinar as asserções através de operadores logicos “and” e “or” e “not”

```
if X /= 0 and Y / X > 1 then …
```

Instruções e controlo

- Análise de casos

```
case A is
   when 0 =>
      Put_Line ("zero");
   when -9 .. -1 | 1 .. 9 =>
      Put_Line ("digit");
   when others =>
      Put_Line ("other")
end case;
```

```
Put_Line (
   (case A is
    when 0 =>
       "zero"
    when -9 .. -1 | 1 .. 9 =>
       "digit"
    when others =>
       "other"));
```

- Deve existir uma cobertura total dos valores que a variável a ser analisada pode assumir, e estes valores devem ser únicos

```
   V : Integer := …;
begin
   case V is
      when 0 =>
         Put_Line (0);
   end case; -- NOK!
```

```
   V : Integer := …;
begin
   case V is
      when 0 =>
         Put_Line ("0");
      when Integer'First .. 0 => -- NOK!
         Put_Line ("Negative");
      when others =>
         null;
   end case;
```

Instruções e controlo - ciclos

- Em Ada dispomos de vários tipos de ciclos através dos quais podemos iterar um número finito de operações
- Os ciclos são declarados, na sua forma mais primitiva como se segue

```
loop
  -- conjunto de instruçoes
end loop;
```

- Dispomos também de forma de controlar a condição de saída de um ciclo:

```
while A < 10 loop
   Put_Line ("Hello");
   A := A + 1;
end loop;
```

- E dispomos também de uma instrução para forçar a saída de um ciclo de forma abrupta

```
loop
   A := A + 1;

   if A > 10 then
      exit;
   end if;

   B := B + 1;
end loop;
```

```
loop
   A := A + 1;
   exit when A > 10;
   B := B + 1;
end loop;
```

Instruções e controlo – ciclos for

- As iterações sobre arrays podem ser declaradas sobre um certo intervalo (definido de maneira ascendente, mas podendo a procura ser invertida)

```
for X in 1 .. 10 loop
   Put_Line ("Hello");
end loop;
```

```
for X in reverse 1 .. 10 loop
   Put_Line ("Hello");
end loop;
```

- O tipo de dados do intervalo pode ser explícito, mas tem que ser necessáriamente um valor discreto; neste caso, se o intervalo não for declarado, é assumido todo o intervalo de valores associados ao tipo em questão

```
for X in Integer range 1 .. 10 loop
   Put_Line ("Hello");
end loop;
```

```
for X in Character loop
   Put_Line (X);
end loop;
```

- A variável usada como iterador num ciclo, é de fato marcada como constante pelo compilador; importante referir que o intervalo de iteração do ciclo é calculado no início do ciclo, e nunca é sujeito a alteração

```
for X in 1 .. 10 loop
   X := X + 1;
end loop;
```

```
for X in A .. B loop
   B := B + 1; -- no effect on the loop
end loop;
```

Tipos compostos - Arrays

- Todos os arrays em Ada são tipados (duplamente: no índice e nos valores). Veja-se o seguinte exemplo:

```
type T is array (Integer range <>) of Integer;

A : T (0 .. 14);
```

- Em Ada, os tipos são definitivos ou não:
  - Definitivo: o seu temanho e restrições são conhecidas;
  - Não-definitivo: são necessárias mais restrições.
  - Os arrays em Ada podem ser de ambas as classes:

```
type Definite is array (Integer range 1 .. 10) of Integer;
type Indefinite is array (Integer range <>) of Integer;

A1 : Definite;
A2 : Indefinite (1 .. 20);
```

- No entanto, o tipo dos valores tem sempre que ser definitivo.

Tipos compostos - Arrays

- Os índices dos arrays podem ser quaisquer tipos discretos, i.e., inteiros e enumerações;
- Podem também ser definidos em função de qualquer intervalo contíguo, se bem que este não é necessário estar sempre presente numa declaração.

```
type A1 is array (Integer range <>) of Integer;
type A2 is array (Character range 'a' .. 'z') of Integer;
type A3 is array (Integer range 1 .. 0) of Integer;
type A4 is array (Boolean) of Integer;
```

- Os componentes dos arrays podem ser acedidos diretamente, e os limites dos seus índices são verificados em tempo de execução:

```
   type A is array (Integer range <>) of Integer;
   V : A (1 .. 10);
begin
   V (1) := 0;
   V (0) := 0; -- NOK
```

- Os componentes dos arrays podem também ser acedidos através do recurso a ciclos:

```
type T is array (Integer range <>) of Integer;

A : T (1 .. 10);

for I in A'Range loop
   A (I) := 0;
end loop;
```

```
type T is array (Integer range <>) of Integer;

A : T (1 .. 10);

for V of A loop
   V := 0;
end loop;
```

Tipos compostos - Arrays

- Podemos ter também arrays bi-dimensionais, ou arrays de arrays:

```
   type T is array (Integer range <>, Integer range <>) of Integer;
   V : T (1 .. 10, 0 .. 2);
begin
   V (1, 0) := 0;
```

```
   type T1 is array (Integer range <>) of Integer;
   type T2 is array (Integer range <>) of T1 (0 .. 2);
   V : T (1 .. 10);
begin
   V (1)(0) := 0;
```

- Sim, o tipo String é, de fato, definido como um array, mas com literais especiais e bibliotecas de suporte:

```
type String is array (Positive range <>) of Character;
```

```
V  : String := "This is it";
V2 : String := "Here come quotes ("")";
```

```
V  : String := "This is nul terminated" & ASCII.NUL;
```

Registos

- Os registos são uma funcionalidade de declaração de tipos complexos, com dados heterogéneos

```
type Shape is record
   Id : Integer;
   X, Y : Float;
end record;
```

```
   S : Shape;
begin
   S.X := 0.0;
   S.Id := 1;
```

```
type Position is record
   X, Y : Integer;
end record;

type Shape is record
   Name : String (1 .. 10);
   P : Position;
end record;
```

- Os registos podem ser inicializados

```
type Position is record
   X : Integer := 0;
   Y : Integer := 0;
end record;
```

- Podem também ser inicializados por nome, aquando da declaração de variáveis

```
P1 : Position := (0, Y => 0);       -- OK
P2 : Position := (X => 0, Y => 0); -- OK
P3 : Position := (Y => 0, X => 0); -- OK
P4 : Position := (X => 0, 0);       -- NOK
```

Registos

- Os registos pode ser parametrizados por um tipo discreto, chamado de descriminante. Este discriminante é visto como uma constante durante a elaboração/declaração das variáveis

```
type Shape_Kind is (Circle, Line, Torus);

type Shape (Kind : Shape_Kind) is record
   X, Y : Float;
   case Kind is
      when Line   =>
         X2, Y2 : Float;
      when Torus  =>
         Outer_Radius, Inner_Radius : Float;
      when Circle    =>
         Radius : Float;
   end case;
end record;
```

```
V : Shape (Circle);
```

```
V1 : Shape := (Kind => Line, X => 0.0, Y => 0.0, X2 => 10.0, Y2 => 10.0);
V2 : Shape := (Circle, 0.0, 0.0, 5.0);
```

Funções e procedimentos (sub-programas)

- Em Ada, existe uma diferenciação entre funções e procedimentos; as primeiras retornam um valor de um determinado tipo, ao passo que as segundas nada retornam (função void em C); Em ambos os tipos de sub-programas, define-se a especificação (assinatura) e implementa-se o corpo

```
function F (V : Integer) return Integer;

procedure P (V : in out Integer);
```

```
function F (V : Integer) return Integer is
   R : Integer := V * 2;
begin
   R := R * 2;
   return R - 1;
end F;
```

- Os parametros dos sub-programas podem ser “in”, “out”, ou “in-out”:
  - in: o valor do parametro não pode ser alterado e é o metodo default;
  - out: o valor do parametro pode ser alterado, e espera-se que o seja, mas também pode ser lido;
  - in-out: espera-se que seja lido e alterado

```
function F (V : in out Integer) return Integer is
   R : Integer := V * 2;
begin
   V := 0;
   R := R * 2;
   return R - 1;
end F;
```

Funções e procedimentos (sub-programas)

- Os parametros podem ser passados por valor ou por cópia para os sub-programas:
  - por cópia implica que o valor passado para o parametro é uma cópia do alvo concreto;
  - por referência, é o próprio alvo que é passado e assim sujeito a alterações
- Se nenhum parametro for definido, então não se usam parentesis (ao contrário de C)

```
function F return Integer;

V : Integer := F;
```

- Podem-se associar nomes aquando da atribuição de valores a parametros de sub-programas

```
procedure P (A, B, C : Integer);

P (B => 0, C => 0, A => 1);
```

- Os parametros "in" podem ter valores por defeito associados (avaliado aquando da chamada do sub-programa)

```
procedure P (A : Integer := 0; B : Integer := 0);
```

```
P;           -- A = 0, B = 0;
P (1);       -- A = 1, B = 0;
P (B => 2); -- A = 0, B = 2;
P (1, 2);    -- A = 1, B = 2;
```

Funções e procedimentos (sub-programas)

- Os sub-programas podem ser embricados, sendo que os sub-programas embricados são visíveis pelos seus “pais” e podem utilizar variáveis declaradas também por estes

```
procedure P (V : Integer) is
   W : Integer;

   procedure Nested is
   begin
      W := V + 1;
   end Nested;
begin
   W := 0;
   Nested;
```

- Funções que consistam apenas na avaliação de uma expressão podem ser declaradas e implementadas através de expressões-função (que serão muito úteis para verificação em SPARK)

```
function Add (L, R : Integer) return Integer is (L + R);
```

- Os sub-programas podem ser overloaded, desde que haja uma diferença nos tipos de parametros e resultado (mas não através de nomes ou modos de acesso “in” vs “out” vs “in-out”)

```
subtype Positive is Integer range 1 .. Integer'Last;
procedure Print (V : Integer);
procedure Print (W : out Positive); -- NOK
```

Packages – declaração

- As packages são um elemento base de engenharia de software em Ada e têm a grande vantagem de forçar a separação entre especificação e implementação

```
-- p.ads

package P is
   procedure Proc;
end P;

-- p.adb

package body P is
   procedure Proc is
   begin
      null;
   end Proc;
end P;
```

```
package P is
   --  parte publica da especificação
   --  declaração de variaveis, suprogramas, tipos, etc...
   --  dados visiveis para o cliente
   --  usado pelo compilador para calcular dependencias
end P;

package body P is
  --  corpo/implementação
  --  declaração de variaveis, suprogramas, tipos, etc...
  --  implementação dos sub-programas
end P;
```

Packages - acesso a componentes

- Apenas as entidades declaradas na parte pública das packages é acessível aos clientes; a referência às entidades é feita através do uso da notação “.”

```
package P1 is

   procedure Pub_Proc;

end P1;

package body P1 is

  procedure Priv_Proc;
  …
end P1;
```

```
package P2 is

   procedure Proc;

end P2;

with P1;

package body P2 is

   procedure Proc is
   begin
      P1.Pub_Proc;
      P1.Priv_Proc;
   end Proc;

end P2;
```

Packages - packages filho

- Uma package filho é uma extensão de uma outra package; as packages filho detêm visibilidade sobre a package pai;
- O propósito de definir estas hierarquias é a de organizar espaços de nomes e partir packages de tamanhos exagerados em blocos mais pequenos

```
-- p.ads
package P is

end P;
```

```
-- p-child_1.ads
package P.Child_1 is

end P.Child_1;
```

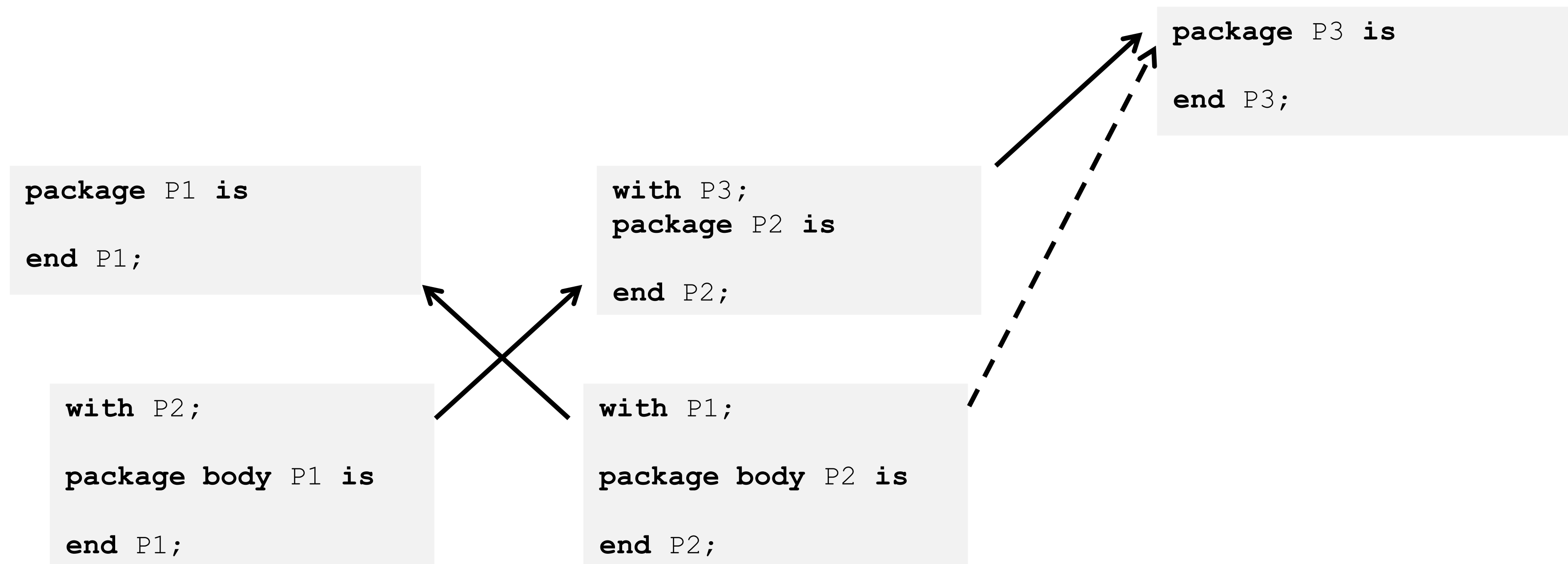
```
-- p-child_2.ads
package P.Child_2 is

end P.Child_2;
```

```
-- p-child_3.ads
package P.Child_3 is

end P.Child_3;
```

```
-- p-child_2-grand_child.ads
package P.Child_2.Grand_Child is

end P.Child_2.Grand_Child;
```

Packages - cláusula “with”

- Tem o propósito de definir dependencias entre packages (para além de outros, não abordados aqui para já...)
- Oferece acesso às declarações publicas de uma package
- Podem ser usadas na especificação e na implementação de packages



Packages - cláusula “use”

- Tem o propósito de evitar o uso de prefixos, mas podem causar ambiguidades
- Necessária para ter acesso aos operadores definidos sobre tipos de dados declarados nas packages de especificação
- Podem ser usadas na especificação e na implementação de packages

```
package P1 is

   procedure Proc1;
   type T is null record;

end P1;
```

```
package P2 is

   procedure Proc1;

end P2;
```

```
with P1;
with P2; use P2;

package body P3 is

   X : T;

   procedure Proc is
      use P1;
      X : T;
   begin
      Proc1;
      P1.Proc1;
      P2.Proc1;
   end Proc;

end P2;
```

Programa exemplo

Vamos ver agora o editor GPS para terem uma ideia...

Parte II – Verificação de Software em Spark

O que são contratos em programas?

Tipos de análise verificação

- Análise de fluxo
- Análise de integridade de código
- Abstrações e contratos
- Verificação funcional de código

Análise de Fluxo

- Modela de variáveis do programa, ou seja:
  - Variáveis globais ou com âmbito definido
  - Variáveis locais
  - Parâmetros formais de outras entidades (sub-programas, tipos, etc.)

- Modela a forma como os dados fluem através das várias instruções do código
  - as variáveis/parâmetros dados como input
  - o resultado final das variáveis

- A estrutura das instruções permite analisar estes modelos e determinar se algo está presente que possa comprometer o comportamento do programa

Análise de Fluxo – inicialização de variáveis

- O GNATProve requer a inicialização das variáveis antes de estas serem lidas
- A análise de fluxo deteta este requisito no código em observação

```
function Max_Array (A : Array_Of_Naturals) return Natural is
  Max : Natural;
begin
  for I in A'Range loop
    if A (I) > Max then    -- Here Max may not be initialized
      Max := A (I);
    end if;
  end loop;
  return Max;
end Max_Array;
```

- Também notifica acerca de instruções “inúteis” e variáveis não usadas

```
procedure Swap1 (X, Y : in out T) is
  Temp : T := X;         -- This variable is unused
  Tmp : T;
begin
  Tmp := X;              -- This statement is ineffective
  X   := Y;
  Y   := X;
end Swap1;
```

Análise de Fluxo – parâmetros formais

- Deteta se os parâmetros de um bloco de código satisfazem os seus requisitos associados

Valor inicial lido	Actualizada num caminho	Atualizada em todos os caminhos	Tipo de parametro
X			**in**
X	X or	X	**in out**
	X		**in out**
		X	**out**

```
procedure Swap (X, Y : in out T) is
  Tmp : T := X;
begin
  Y := X;    -- The initial value of Y is not used
  X := Tmp;  -- Y is computed to be out
end Swap;
```

Análise de Fluxo – contratos globais

- Contratos referentes às associações de conteúdo global no código
  - Podem ser “in”, “out”, ou “in_out”;
  - a keyword “null” pode ser usada para declarar a não interferência com variáveis globais

```
procedure Set_X_To_Y_Plus_Z with
  Global => (Input  => (Y, Z), -- reads values of Y and Z
             Output => X);     -- modifies value of X

procedure Set_X_To_X_Plus_Y with
  Global => (Input  => Y,  -- reads value of Y
             In_Out => X); -- modifies value of X
                           -- also reads its initial value

function Get_Value_Of_X return Natural with
  Global => X;  -- reads the value of the global variable X

procedure Incr_Parameter_X (X : in out Natural) with
  Global => null; -- do not reference any global variable
```

Análise de Fluxo – contratos de dependências

- Anunciam as dependencias entre inputs e outputs de sub-programas
  - Considera parametros e variabeis globais
- O GNATprove verifica estas dependencias contra os corpos dos sub-programas

```
procedure Swap (X, Y : in out T) with
  Depends => (X => Y,              -- X depends on the initial value of Y
              Y => X);             -- Y depends on the initial value of X

function Get_Value_Of_X return Natural with
  Depends => (Get_Value_Of_X'Result => X);    -- result depends on X

procedure Set_X_To_Y_Plus_Z with
  Depends => (X => (Y, Z));        -- X depends on Y and Z

procedure Set_X_To_X_Plus_Y with
  Depends => (X =>+ Y);            -- X depends on Y and X's initial value

procedure Do_Nothing (X : T) with
  Depends => (null => X);          -- No output is affected by X

procedure Set_X_To_Zero with
  Depends => (X => null);          -- X depends on no input
```

Análise de integridade

- Consiste em identificar e alertar para a possível ocorrência de erros em tempo de execução
- A integridade pode ser garantida quer estaticamente, ou através da transformação destes em asserções lógicas (pelo compilador) que são verificadas em tempo de execução

```
type Nat_Array is array (Integer range <>) of Natural;

A : Nat_Array (1 .. 10);
I, J, P, Q : Integer;

A (I + J) := P / Q; -- potentially problematic

A (Integer'Last + 1) := P / Q;
raised CONSTRAINT_ERROR : overflow check failed

A (A'Last + 1) := P / Q;
raised CONSTRAINT_ERROR : index check failed

A (I + J) := Integer'First / (-1);
raised CONSTRAINT_ERROR : overflow check failed

A (I + J) := 1 / (-1);
raised CONSTRAINT_ERROR : range check failed

A (I + J) := P / 0;
raised CONSTRAINT_ERROR : divide by zero
```

Análise de integridade - modularidade

- O GNATProve usa os contratos/aspetos para fazer demonstrações sobre o program de forma modular, tendo como unidade base sub-programas:
  - Quando analisa um sub-programa, a pré-condição é tudo o que é conhecido sobre os dados de entrada
  - Quando o sub-programa é chamado, a sua pós-condição é tudo o que é conhecido sobre os dados de saída

```
procedure Increment (X : in out Integer) with
  Pre => X < Integer'Last is
begin
  X := X + 1;
-- info: overflow check proved
end;
```

```
X := Integer'Last - 2;
Increment (X);
-- Here GNATprove does not know the value of X
X := X + 1;
-- medium: overflow check might fail
```

Análise de integridade – verificação em tempo de execução

- Os contratos pertencentes ao Ada 2012 (o Spark suporta apenas parte da totalidade) podem ser verificados em tempo de execução
  - Contratos são transformados em asserções lógicas que são invocadas em tempo de execução; estas são introduzidas pelo compilador
  - Se um contrato falhar durante a execução, o programa lança uma exceção

```
procedure Increment (X : in out Integer) with
  Pre => X < Integer'Last;
X := Integer'Last;
Increment (X);
-- raised ASSERT_FAILURE : failed precondition

procedure Absolute (X : in out Integer) with
  Post => X >= 0 is
begin
  if X > 0 then
    X := - X;
  end if;
end Absolute;
X := 1;
Absolute (X);
-- raised ASSERT_FAILURE : failed postcondition
```

Análise de integridade – verificação estática

- O GNATProve tenta demonstrar, estáticamente, todas as pré- e pós-condições
  - As pré-condições são verificadas em cada chamada de um sub-programa
  - As pós-condições são verificadas uma só vez como sendo parte do corpo (implementação) do sub-programa

```
procedure Increment (X : in out Integer) with
  Pre => X < Integer'Last;
X := Integer'Last;
Increment (X);
-- medium: precondition might fail

procedure Absolute (X : in out Integer) with
  Post => X >= 0 is
-- medium: postcondition might fail, requires X >= 0
begin
  if X > 0 then
    X := - X;
  end if;
end Absolute;
X := 1;
Absolute (X);
```

Análise de integridade – outros tipos de contratos

- Permite a introdução de assunções sobre o código, e estas são adicionadas ao contexto lógico para efeitos de contruir a demonstração de correção
- Os “contract-cases” permitem anotar o sub-programa como um conjunto disjunto de casos a validar

```
procedure Absolute (X : in out Integer) with
  Pre            => X > Integer’First,
  Contract_Cases => (X < 0  => X = - X’Old,
                     X >= 0 => X = X’Old);
-- info: disjoint contract cases proved
-- info: complete contract cases proved
-- info: contract case proved

pragma Assume (X < Integer’Last);
X := X + 1;
```

Abstrações e contratos

- Conceito base é o de ter diferentes visões sobre o mesmo objeto
  - Uma abstração captura o que faz
  - Um refinamento fornece informação concreta e detalhada sobre como é que o faz

```
procedure Increase (X : in out Integer) with
  Global => null,
  Pre    => X <= 100,
  Post   => X'Old < X;

procedure Increase (X : in out Integer) is
begin
  X := X + 1;
end Increase;
```

- Assim, uma especificação sumariza aquilo em que podemos confiar e assumir
- Um abstração simplifica a implementação e verificação (os utilizadores só precisam de saber o comportamento associado, e não os detalhes)
- As abstrações simplificam a manutenção do código, pois modificações na implementação do objeto não afetam a sua utilização pelos clientes

Abstrações e contratos – abstração de um estado

- As variáveis declaradas numa package fazem parte do seu estado
- O estado de uma package pode ser ou não visível
  - No caso de serem públicas, fazem parte da especificação
  - No caso de serem privadas, permitem abstração

```
package Stack is
  procedure Pop  (E : out Element);
  procedure Push (E : in  Element);
end Stack;

package body Stack is
  Content : Element_Array (1 .. Max);
  Top     : Natural;
```

```
package Stack with
  Abstract_State => The_Stack
is
  ...

package Stack with
  Abstract_State => (Top_State, Content_State)
is
  ...

pragma Assert (Stack.Top_State = ...);
--  Compilation error: Top_State is not a variable
```

- Cada abstração tem que ser refinada na implementação

```
package body Stack with
  Refined_State => (The_Stack => (Content, Top))
is
  Content : Element_Array (1 .. Max);
  Top     : Natural;
  --  Both Content and Top must be listed in the list of constituents of The_Stack
```

Abstrações e contratos – variáveis privadas

- A parte privada de uma especificação pode ser visível enquanto que o seu corpo/implementação não
- Uma package com uma abstração, as variaveis privadas têm que ser associadas com essa abstração de estado no momento da sua declaração

```
package Stack with Abstract_State => The_Stack is

  procedure Pop  (E : out Element);
  procedure Push (E : in  Element);

private
  Content : Element_Array (...) with Part_Of => The_Stack;
  Top     : Natural             with Part_Of => The_Stack;
end Stack;

package body Stack with
  Refined_State => (The_Stack => (Content, Top))
```

Abstrações e contratos – sub-programas

- O refinamento de pré- e pós-condições é obtido através de “expression-functions”:
  - No mesmo package, o corpo da “expression-function” é usado para verificação
  - Fora do package, a ”expression-function” é uma caixa-nega
- Pode-se usar o contrato “Refined_Post” para fortalece a pós condição

```
package Stack
  ...
  function Is_Empty return Boolean;
  function Is_Full  return Boolean;

  procedure Push (E : Element) with
    Pre  => not Is_Full,
    Post => not Is_Empty;

package body Stack
  ...
  function Is_Empty return Boolean is (Top = 0);
  function Is_Full  return Boolean is (Top = Max);

  procedure Push (E : Element) with
    Refined_Post => not Is_Empty and E = Content (Top);
```

Abstrações e contratos – inicialização de variáveis locais

- O contrato “Initializes” permite especificar variáveis inicializadas durante a elaboração de uma package:
  - É opcional, mas o compilador calcula uma aproximação do conjunto variáveis locais a serem inicializadas
  - Se usado explicitamente, tem que listar todos constituintes do estado (publicos ou não) que são inicializados

```
package Stack with
  Abstract_State => The_Stack,
  Initializes    => The_Stack
is
--  Flow analysis will make sure both Top and Content are
--  initialized at package elaboration
```

- Se o valor inicial da variável depender de um valor externo, então tal tem que ser explicito no contrato

```
package P with
  Initializes => (V1, V2 => External_Variable)
is
  V1 : Integer := 0;
  V2 : Integer := External_Variable;
end P;

--  The association for V1 is omitted, it does not depend
--  on any external state.
```

Correção funcional

- Uma demonstração de correção funcional dá garantias totais de que o programa se comporta de acordo com a sua especificação
- De forma a verificar a correção, em Spark temos que recorrer a contratos

```
function Find (A : Nat_Array; E : Natural) return Natural
with Post => Find'Result in 0 | A'Range;

function Find (A : Nat_Array; E : Natural) return Natural
with Post =>
  (if (for all I in A'Range => A (I) /= E)
   then Find'Result = 0
   else Find'Result in A'Range and then A (Find'Result) = E);
```

- O Ada 2012 permite exprimir contratos poderosos, oferencendo sintaxe com noção de quantificadores, analise de casos, expression functions, predicatos sobre tipos, invariantes locais, etc.

```
function Is_Sorted (A : Nat_Array) return Boolean is
  (for all I in A'Range =>
    (if I < A'Last then A (I) <= A (I + 1)));
--  Returns True if A is sorted in increasing order.

subtype Sorted_Nat_Array is Nat_Array with
  Dynamic_Predicate => Is_Sorted (Sorted_Nat_Array);
--  Elements of type Sorted_Nat_Array are all sorted.
```

Correção funcional – código fantasma

- Código fantasmas é um sub-conjunto do código normal de Ada 2012 que é usado apenas para especificação
  - Não pode ter qualquer efeito sobre o comportamento do programa
  - Se for usado em asserções lógicas (pragma Assert), então é executado como código normal
  - Pode-se instruir o computador para não gerar código ghost (situação ideal)

```
procedure Do_Something (X : in out T) is
  X_Init : constant T := X with Ghost;
begin
  Do_Some_Complex_Stuff (X);
  pragma Assert (Is_Correct (X_Init, X));
  --  It is OK to use X_Init inside an assertion.

  X := X_Init;
  --  Compilation error:
  --     Ghost entity cannot appear in this context.
```

- Funções marcadas como “ghost” só podem ser usadas em contratos

```
type Stack is private;

function Get_Model (S : Stack) return Nat_Array with Ghost;

procedure Push (S : in out Stack; E : Natural) with
  Pre  => Get_Model (S)'Length < Max,
  Post => Get_Model (S) = Get_Model (S)'Old & E;

function Peek (S : Stack; I : Positive) return Natural is
  (Get_Model (S) (I));
-- Get_Model cannot be used in this context.
```

Correção funcional – invariantes de ciclo

- As invariantes de ciclo são contratos usados para marcar fases na verificação de ciclos
  - A invariante é verificada na primeira iteração do ciclo
  - O corpo do ciclo, a preservação da invariante e as instruções do código do ciclo são todas verificadas assumindo que a invariante se verificou na iteração anterior (argumento de indução...)

```
function Find (A : Nat_Array; E : Natural) return Natural is
begin
  for I in A'Range loop
    pragma Loop_Invariant
          (for all J in A'First .. I - 1 => A (J) /= E);
    --  info: loop invariant initialization proved
    --  info: loop invariant preservation proved
    if A (I) = E then
      return I;
    end if;
  end loop;
  pragma Assert (for all I in A'Range => A (I) /= E);
  --  info: assertion proved
```

Correção funcional – invariantes de ciclo (boas praticas)

- Na prática, é muito relevante satisfazer os seguintes passos para garantir a correção de uma invariante de ciclo:
  - Garantir que é verificada na primeira iteração (condição booleana e variáveis a que se refere)
  - Deve permitir demonstrar a ausência de erros em tempo de execução e asserções locais
  - Deve permitir demonstrar a pós condição de saída do ciclo
  - Deve ser preservada em quaisquer duas iterações consecutivas do ciclo

```
A_I : constant Nat_Array := A with Ghost;
for K in A'Range loop
 A (K) := F (A (K));
 pragma Loop_Invariant
      (for all J in A'First .. K => A (J) = F (A_I (J)));
  --  info: loop invariant initialization proved
  --  medium: loop invariant might fail after first iteration
end loop;
pragma Assert (for all K in A'Range => A (K) = F (A_I (K)));
--  info: assertion proved
```