Programming Languages

G22.2110
Ada, continued
The Ada type system: types and subtypes

- a type consists of a set of values and a group of operations
- a type declaration specifies static properties of objects
- a subtype declaration specifies dynamic properties of values.
- Ada uses name equivalence instead of structural equivalence:
  - types with different names are distinct and incompatible
- subtypes of the same base type are compatible.
Compile-time vs. run-time

Type properties are enforced by the compiler:

A: Integer := False; -- compile-time error
B: Positive := F (Z); -- run-time check
     -- (assuming F returns an Integer)
Scalar Types Overview

- discrete types
  - integer types
  - enumeration types

- real types
  - floating-point types
  - fixed-point types
Predefined Scalar Types

Several predefined types:

    Integer, Long_Integer, Short_Integer

Specific bounds of type must be static:

    type My_Int is range -2 ** 16 .. 2 ** 16 -1;
    type Tiny is range 0 .. 10;

Explicit bounds make program more portable: compiler will figure out the corresponding hardware type.

Modular types:

    type Byte is mod 2 ** 8;
Integer Operations

- comparison operators: = /= < <= > >=
- addition operators: − +
- unary operators: − +
- multiplying operators: * / mod rem
- highest precedence operators: ** abs
Boolean Operations

- boolean binary operators: `and`, `or`, `xor`
- boolean unary operators: `not`
- short-circuit operations: `and then`, `or else`
- membership operations: `in`, `not in`

When in doubt, parenthesize!
type Integer is ... -- implementation defined

subtype Positive is Integer range 1 .. Integer’Last;
subtype Natural is Integer range 0 .. Integer’Last;

X: Integer := 500;
Y: Positive := 2 * X;
Z: Natural := -Y;   -- raises constraint error
Enumeration types: abstraction at its best

- trivial implementation: literals are mapped to successive integers
- very common abstraction: list of names, properties
- expressive of real-world domain, hides machine representation

Examples:

```pascal
  type Suit is (Hearts, Diamonds, Spades, Clubs);
  type Direction is (East, West, North, South, Lost);
```

Order of list means that Spades > Hearts, etc.

Contrast this with C#:

“arithmetics on enum numbers may produce results in the underlying representation type that do not correspond to any declared enum member; this is not an error”
type Fruit is (Apple, Orange, Grape, Apricot);
type Vendor is (Apple, IBM, HP, Dell);

My_PC : Vendor;
Dessert : Fruit;
...
My_PC := Apple;
Dessert := Apple;
Dessert := My_PC;  -- error
type Boolean is (False, True);

type Character is (...);
  -- full ASCII (Ada83) or Latin_1 (Ada95)
  -- not expressible in Ada

type Wide_Character is (...);  -- Unicode or ISO646
Real Types

All computations are approximate:

- Fixed point type: absolute bound on error:
  
  ```
  type Temp is delta 2 ** (-16) range -100.0 .. 100.0;
  ```

- Floating point type: relative bound on error:
  
  ```
  type Angle is digits 7 range -2.0 .. 2.0;
  ```

Predefined floating point types: Float, Long_Float, etc.
Array Types

Index type(s) of an array type are typed:

```pascal
  type Weekday is (Mon, Tue, Wed, Thu, Fri);

  type Workhours is array (Weekday) of Integer;
  type Pressure is array (1..1000,
                        1..1000,
                        1..1000) of Long_Float;

  type String is array (Positive range <>) of Character;
```
Conventional named fields:

```haskell
type Buffer is record
    size: Positive;
    contents: String (1 .. 100);
end record;

B1: Buffer;
    -- can use B1, B1.size, B1.contents(10), ...
```
Variants

Need to treat group of related representations as a single type:

```ada
type Figure_Kind is (Circle, Square, Line);

type Figure (Kind: Figure_Kind) is record
  Color: Color_Type;
  Visible: Boolean;
  case Kind is
    when Line => Length: Integer;
      Orientation: Float;
      Start: Point;
    when Square =>
      Lower_Left, Upper_Right: Point;
    when Circle =>
      Radius: Integer;
      Center: Point;
  end case;
end record;
```
C1: Figure (Circle); -- discriminant provides constraint
S1: Figure (Square);
...
C1.Radius := 15;
if S1.Lower_Left = C1.Center then ...

function Area (F : Figure) return Float is
   -- applies to any figure, i.e. subtype
begin
   case F.Kind is
      when Circle => return Pi * Radius ** 2;
      ...
   end Area;
Discriminant checking, part 2

L : Figure (Line);
F : Figure;               -- illegal, don’t know which
P1 := Point;
...
C := (Circle, Red, False, 10, P1);
   -- record aggregate
... C.Orientation ...
   -- illegal, circles have no orientation
C := L;
   -- illegal, different kinds
C.Kind := Square;
   -- illegal, discriminant is constant

Discriminant is a visible constant component of object.
There is a way of specifying a figure that can change kinds
Access Types

Typed pointers, for type safety and to minimize aliasing:

```haskell
type Handle is access Buffer;
Ptr: Handle := new Buffer;

Ptr.all is a Buffer.

can write Ptr.size, Ptr.contents, etc.
```
Derived Types

A general mechanism for creating new types with the properties of existing ones:

```pascal
type Like_T is new T;  -- same set of values, same

type Small_Int is range 1 .. 10;
```

equivalent to

```pascal
type Anon is new Integer;
subtype Small_Int is Anon range 1 .. 10;
```

and all arithmetic operations are inherited
Blocks

```
declare
  X: Integer := F (5);  
  Y: Integer := 2 * X;  
  Z: Integer := Y * Z;  
  X: Float;  
begin
  declare
    X: Float := Float (Y);  
  begin
    Put_Line (Float'Image (X));
  end;
end;
```
Variables and Constants

Variable declarations:

Limit: Integer := 25;
Offset: Integer range 1 .. 20;

Constant declaration:

Sqrt2: constant Float := Sqrt (2.0);
    -- not static

Always: constant Boolean := True;
    -- static value

Sometimes: constant Boolean := not Always;
    -- static expression
Variables must be constrained

Subtype is constrained:

First_Name: String (1..5) := "Ralph";

but not necessarily static:

Last_Name: String (1 .. X * 2);

else subtype is indefinite but initial value provides bounds:

Comment: String := "this_is_obvious";
    -- bounds are 1 .. 15
Multiple Declarations

This, That: T := F (1, 2, 3);

Is equivalent to

This: T := F (1, 2, 3);
That: T := F (1, 2, 3);

F is called twice. *Important if expression has side-effect:*

```plaintext
    type Ptr is access R;
    P1, P2: Ptr := new R;
```

two R’s are allocated!
Pi: constant := 3.14159265;
    -- type inferred from value

Half_Pi: constant := Pi / 2;
    -- mixed arithmetic OK

Big: constant := 2 ** 200;
    -- legal

One: constant := 2 * Big / (Big + Big);
    -- must be exact
Attributes

- attributes denote properties of a type, or type-specific properties of a value

  Boolean’Siz e  --  1, because single bit
  Character’Siz e  --  8 bits
  Month’Pos (Jul)  --  involves type and literal
  Table’Length (1)  --  specify array and dimension

- could be written as functions, except that functions don’t take types as arguments; hence, new syntax.
Attributes on Discrete Types

Byte'First, Long_Integer'Last
   -- applies to type or subtype
Weekday'Succ (Today)
   -- like function call
Integer'Succ (X*Y)    -- like adding one
Boolean'Pred (True)  -- yields False
Boolean'Succ (True)  -- exception
Weekday'Pos (Mon)    -- yields 0
Weekday'Val (3)      -- yields Thu
Positive'Max (X, Y)
   -- function with two args
Array Types

Index types can be of any discrete type

Component type must be definite:

```plsql
type class_list is array (1 .. 100) of String (1..10);

-- OK

type class_list is array (1 .. 100) of String;

-- Error
```

Subtype constrains all indices or none:

```plsql

subtype Table is Matrix;

subtype Rotation is Matrix (1 .. 3, 1 .. 3);
```

Arrays can be assigned as a whole:

```plsql
Table1 := Table2;  -- all components assigned
```
Anonymous Array Types

Grades: $\text{array } (1 \ldots \text{Num}\_\text{Students}) \text{ of Natural;}$

Type of Grades has no name $\Rightarrow$ distinct from any other array types.

$\text{Ar1, Ar2 : array } (1 \ldots 10) \text{ of Boolean;}$

$\ldots$

$\text{Ar1 := Ar2;}$   -- Error: different (anonymous) types

Moral: If a type is useful, it deserves to have a name.
Array Attributes

type Matrix is array (Positive range <>, Positive range <>);
subtype Rect is Matrix (1 .. 3, 1 .. 5);
M3: Rect;

M3’First (1)   --yields 1
M3’First       --same
Rect’length (2) --yields 5 (applies to type)
M3’range (2)   --equivalent to 1 .. 5
String’Length  --ERROR: unconstrained

Arrays are self-describing; size information is “built-in”
Array Aggregates

Expression that yields an array value:

A := (1, 2, 3, 10);                      --positional
A := (1, others => 0);                   --for default
A := (1..3 => 1, 4 => -999);            --named

Default can only be used if bounds are known:

A: String(1 .. 10) := (others => '?');   --OK
A: String             := (others => '?'); --Error: un.
Initializer in C++

Similar notion for declarations:

```cpp
int v2[] = { 1, 2, 3, 4 }; // size from initializer
char v3[2] = { 'a', 'z' }; // declared size
int v5[10] = { -1 }; // default: other components
char name[] = "Algol"; // string literals are aggregate
```

but no array assignments, so initializer is not an expression
(mechanism is less orthogonal)
Aggregates and Qualification

Aggregate may be ambiguous:

```pascal
type Vector is array (1 .. 3) of Float;
procedure Display (V : Vector);

type Assay is array (1 .. 3) of Float;
procedure Display (A : Assay);

Display ((1.0, 1.2, 1.5)); --ambiguous
Display (Vector'(1.0, 1.2, 1.5)); --OK
```
Multidimensional Arrays

Aggregates given in row-major order with subaggregates:

```pascal
type Square is array (1 .. 3, 1 .. 3) of Integer;
Unit: constant Square := ((1, 0 ,0), (0, 1, 0), (0, 0, 1));
```

A two-dimensional array is *not* an array of arrays:

```pascal
type Vector is array (1 .. 3) of Integer;
type V3 is array (1 .. 3) of Vector;

V3 is not convertible to Square.
```
Operations on One-Dimensional Arrays

Boolean operations extend pointwise:

```pascal
type Set is array (1 .. Card) of Boolean;
S1, S2, S3: Set;
...
S3 := S1 and S2; -- Intersection
```

Lexicographic comparisons on arrays of discrete types:

```pascal
S1 := (T, T, T);
S2 := (T, T, F);
...
S2 < S1 -- yields True
```
Concatenation and Slicing

Both operations yield the base type:

```pascal
type Table is array (1..10) of Integer;
T1, T2: Table;
...
T1 & T2  -- What type?
```

Declaration equivalent to:

```pascal
type Anon is array (Integer range <>) of Integer;
subtype Table is Anon (1 .. 10);
  -- T1 & T2, T1 (X .. Y) are of type Anon
```
Specifying a range

**subtype** Sub **is** Positive **range** 2 .. 4;
Label: String (1..10) := "transcends";
...

Label (2 .. 4) -- Yields "ran"
Label (Integer **range** 2 .. 4) -- Same
Label (Sub) -- Ditto

Also used in loops and case statements.
Goto Statement

Occasionally useful.

Syntax of maximum ugliness to discourage use:

```plaintext
while Going loop
  Compute_Some;
  if Hopeless then goto next_try; end if;
  Compute_Some_More;
  <<next_try>> Adjust_Computation;
end loop;
```

Restricted range.
Subprograms

- two types: functions and procedures
- functions:
  - only have in-parameters
  - may return unconstrained types
  - execution must end in return statement
- parameter passing:
  - by copy-return for scalar types
  - not specified for non-tagged types
- positional and named notation in calls
- defaults for in-parameters
Functions

function F (X: Integer := 0;
    Y: Float := Pi;
    Maybe: Boolean := True) return Integer;
...
F (10, 0.0, False)
F (5)     -- F (5, Pi, True)
F (Maybe => False)     -- F (0, Pi, False)
Operators

Like functions, but usable in infix notation.

```pascal
package Bignums is
    type Bignum is private;
    Zero: constant Bignum;    -- deferred constant

    function "+" (X, Y: Bignum) return Bignum;
    function "*" (X, Y: Bignum) return Bignum;
    function Image (X: Bignum) return String;

    private ...
End Bignums;
```

Must respect syntax: no defaults, no unary "*", etc.
Abstraction mechanisms

- packages
- private types
- objects and inheritance
- classes, polymorphism, dynamic dispatching
- generic units
- concurrency: tasks and protected types
Packages

- a related set of types, constants, and subprograms
- separate declaration (interface) and implementation
- supports privacy and data hiding
- the single most important idea in software engineering
A package for stacks

package Stacks is

    type Stack is private;

    procedure Push (C: Character;
        On: in out Stack);

    procedure Pop (C: Character;
        From: in out Stack);

    function Empty (S: Stack) return Boolean;

private

    type Stack is record
        Top: Integer := 0;
        Contents: String (1..80) := (others => "*");
    end record;

end Stacks;
Object-oriented Programming

- type extension
- inheritance and overriding
- run-time dispatching
- polymorphism
- class-wide programming
- abstract types and subprograms
Type Extension

Mechanism to define new types by enrichment:

```pascal
type Point is tagged record
    X_Coord, Y_Coord : Integer;
end record;

type Pixel is new Point with record
    R, G, B : Integer;
end record;
```
Inheritance

- a type has *primitive operations*: operations that involve the type as a parameter or a returned value.
- a type extension inherits the primitive operations of its parent.
- a primitive operation can be redefined and overridden for a descendant type.
Polymorphism

- a class is a family of types with the same ancestor.
- an object of the class is identified at run-time by its tag.
- dynamic dispatching uses the tag of the object to determine the operation that applies.
- a classwide operation applies uniformly to all members of the class:

  ```
  procedure Examine (Thing: in out T'Class);
  ```
Generic Units

- basic tool for software reuse
- parameters can be types, objects, subprograms, packages
- similar to C++ templates
- absence from Java is mind-boggling
A Generic Package

generic

  type  T  is  private;

package  Stacks  is

  type  Stack  is  private;

  procedure  Push  (Thing:  T;  On:  in  out  Stack);

  ...

private

  type  Arr  is  array  (1..100)  of  T;

  type  Stack  is  record
  
    Top:  Integer  :=  100;

    Contents:  Arr;
  
  end  record;

end  Stacks;
A Generic Subprogram

generic

    type T is private;
    type Arr is array (Integer range <>) of T;
    with function "<" (X, Y: T) return Boolean;

procedure Sort (Table: in out Arr);
Declarations and Scope

- declarations are elaborated in order
- entities become visible at the end of their declaration (usually)
- block structure allows arbitrary nesting of declarative regions
- declarations can appear in
  - subprograms
  - packages
  - blocks