The Ada programming language

An introduction for TTK4145 – Lecture 2

Kristoffer Nyborg Gregertsen

SINTEF Digital
Department of Mathematics and Cybernetics

2018-02-16

Contents

Contents

What is Ada?

- Ada is a general purpose programming language designed for safety and maintainability
- ISO standard versions Ada 83, Ada 95, Ada 2005
- Ada 2012 is the latest revision of the standard
- Ada has built-in language support for tasking and a rich set of synchronization primitives
- Ada is a mature language with excellent tool support on many platforms

Hello world!

```
with Ada.Text_IO;

procedure Hello_World is
begin
   Ada.Text_IO.Put_Line ("Hello␣World!");
end Hello_World;
```

- File called “hello_world.adb”
- Compile with: `gnatmake hello_world`

Variables and constants

- Naming convention for all identifiers is Like_This
- Ada code is **not** case-sensitive
- Variables are declared with name first, then the type
- Constants may be of a type or just a named number
- Named numbers are of universal type without limits in size or precision
- Constants of a type are like variables only ... *constant*

Scope

- Variables, types, routines, tasks and more all have a scope
- Defined in statement section by **declare – begin – end**
- Also defined by language constructs such as routines and task bodies

```ada
procedure My_Procedure is
   X : Float := -1.0;          -- Declarations
begin

   X := X ** 2;                -- Statements, no declarations

   declare
      Y : constant := 0.1;     -- More declarations
   begin
      X := X + Y;              -- Statements
   end;

end My_Procedure;
```

Data-types

- Ada is a strongly typed language
- No implicit type-casting as in C
- Two primary classes of types:
  - Primitives
  - Composite
- The primitive types are sub-divided in:
  - Scalars
  - References

Scalars

- Discrete scalars:
  - Enumeration types such as: Boolean
  - Integer types such as: Integer, Natural, Positive
- Real scalars:
  - Float types
  - Fixed types

References

- May define access types for any type and routine
- Dereferenced using the **all** operator
- Need no explicit dereference when unambiguous
- Heap memory allocated with the **new** operator
- No garbage collector, need explicit deallocation!
- References are less used in Ada than in C

Composite

- Composite types may contain:
  - Primitives
  - Other composite types
- Five classes of composite types:
  - **array**
  - **record**
  - **interface**
  - **protected**
  - **task**
- Today we will talk about protected and task types too!

Program flow and exceptions

- Ada supports standard program flow constructs:
  - **if** ... **then** ... **elsif** ... **else**
  - **case**
  - **loop**
  - **for**
  - **while**
  - **goto** (!!!)
- Exceptions are used for error handling and are either handled before **end** or propagate downward on call stack
- Program or task halts when exception reaches bottom of stack

Routines

- There are two types of routines:
  - Procedures without return value
  - Functions with return value
- Functions *should* not have side effects
- No empty () for routines without arguments
- Routines may have default values for arguments

Packages

- The building blocks of Ada applications
- Two parts the specification (.ads) and body (.adb):
- Specification has a public and a private section
- Body contains implementation of routines
- The body may also have internal declarations and routines
- A task can live in a package scope, initialized with package

Object-orientation

- Similar OO-model as Java:
  - Classes
  - Interfaces
- OO-model based on **tagged** records
- The definition of a class and its methods are usually gathered in a package, no link between class and file as in Java
- Abstract classes may have abstract and null methods
- For interfaces only abstract and null methods are allowed
- Dispatching calls only for class-wide types (Type'Class)
- Interfaces can also be used for concurrent constructs!

Concurrent constructs

- Ada has rich built-in support for tasking and synchronization
  - Task and protected object types
  - Task rendezvous
  - Protected entries
  - Asynchronous control
- Real-time specifics in Annex D of standard discussed later
- Programs with tasks are easy to write compared to C/POSIX
- Multitasking programs are portable from PC to embedded!

Single tasks

- A single task may be created using keyword **task**
- Need package Ada.Real_Time for Time, Clock and Milliseconds
- The task has default priority since none is given

```
task Periodic;

task body Periodic is
   Next : Time := Clock;
begin
   loop
      delay until Next;
      ...
      Next := Next + Milliseconds (100);
   end loop;
end Periodic;
```

Task type

- A task types allow several similar task instances to be created
- May give a primitive argument called a *discriminant* in Ada

```
task type Worker (N : Character);

task body Worker is
begin
    Put_Line ("My␣name␣is␣" & N);
    ...
end Worker;

A : Worker ('A');
B : Worker ('B');
```

Execution

- Tasks are *ready* for execution when they enter scope, which task starts executing depends on scheduling
- If the tasks are in local scope, the creating task cannot leave this scope before the tasks have terminated
- Tasks that are created on library level (within a package) live for the entire execution of the program
- Tasks may also be created on heap using the **new** command

Communication

- Tasks may communicate and synchronize:
  - Synchronously through task *rendezvous*
  - Asynchronously through protected objects
- For synchronous communication a task may:
  - Have several entries used for rendezvous
  - Block waiting for several entries using **select**
  - Have a timeout when waiting on a entry
  - Have an immediate alternative if no entry is ready
- Protected objects are discussed later

Example

```
task type Runner is
   entry Start; -- One entry, no arguments
end Runner;

task body Runner is
begin
   accept Start; -- Block here
   ...           -- Do something
end Runner;

declare
   A, B : Runner;
begin
   A.Start;      -- Start A first
   delay 1.0;
   B.Start;      -- Start B one second later
end;             -- Block here until A and B are done
```

Example

```
task type Server (S : Integer) is
   entry Write (I : Integer);
   entry Read  (I : out Integer);
end Server;

task body Server is
   N : Integer := S;
begin
   loop
      select
         accept Write (I : Integer) do
            N := I;
         end;
      or
         accept Read (I : out Integer) do
            I := N;
         end;
      end select;
   end loop;
end Server;
```

Timeout and immediate alternative

```
select
   accept Signal;
   ...  -- Do this if a task calls Signal within one second
or
   delay 1.0;
   ..   -- Else do this
end select;

select
   accept Signal;
   ..   -- Do this if a task is already blocked on Signal
else
   ...  -- Else do this immediately (same as zero timeout)
end select;
```

Protected objects

- Special composite type used for synchronization
- May have a single protected object or class of objects:
  - **protected** Name
  - **protected type** Name
- Protected objects may have:
  - Entries with a guard – may block calling tasks
  - Procedures for exclusive access to internal data
  - Functions for reading internal data (read-only)
- Entries are open or locked depending on the Boolean guard
- Calling tasks are queued on an entry (usually FIFO)

Example

- Protected object implementing a counting semaphore
- Uncommon to implement low-level semaphore using high-level protected object, normally other way around
- Done here since semaphore has well known behavior
- Notice the private part of the protected object, this part may also contain entries, procedures and functions for internal use

```
protected type Semaphore (N : Positive) is
   entry Lock;
   procedure Unlock;
   function Value return Natural;
private
   V : Natural := N;
end Semaphore;
```

Example

```
protected body Semaphore is

   entry Lock when V > 0 is
   begin
      V := V – 1;
   end Lock;

   procedure Unlock is
   begin
      V := V + 1;
   end Unlock;

   function Value return Natural is
   begin
      return V;
   end Value;

end Semaphore;
```

Example

```
task type Worker (Mutex : not null access Semaphore);

task body Worker is
begin
   Mutex.Lock;
   Put ("Starting... ");
   delay 1.0;
   Put_Line ("Done!");
   Mutex.Unlock;
end Worker;

declare
   Mutex : aliased Semaphore (1);
   A, B, C : Worker (Mutex'Access);
begin
   null;
end;
```

Advanced features

- Possible to get the number of tasks blocked on an entry using
  Entry_Name'Count
- Possible to move a task to the queue of another entry using
  requeue Entry_Name
- To requeue the other entry must have the same arguments or none
- It is possible to have families of entries, i.e. for priority
- A protected procedure may be used as interrupt handler
- A protected object with an interrupt handler must be at library level, that is, in a package

Example

```
pragma Unreserve_All_Interrupts;

protected Terminator is

   entry Wait_Termination;

private

   entry Wait_Final;
   procedure Ctrl_C;
   pragma Attach_Handler (Ctrl_C, SIGINT);

   Count : Natural := 0;
   Final : Boolean := False;

end Terminator;
```

Example

```
protected body Terminator is

   entry Wait_Termination when Count > 0 is
   begin
      Count := Count - 1;
      requeue Wait_Final;
   end Wait_Termination;

   entry Wait_Final when Final is
   begin
      Ada.Text_IO.Put_Line ("Hasta␣la␣vista,␣baby!");
   end Wait_Final;

   procedure Ctrl_C is
   begin
      Count := Wait_Termination'Count;
      Final := Wait_Final'Count > 0;
   end Ctrl_C;

end Terminator;
```

Example

```
type Priority is (High, Low);

task Worker is
  entry Handle (Priority)(J : Job);
end Worker;

task body Worker is
begin
  loop
    select
      accept Handle (High)(J : Job) do
        ...
      end;
    or
      when Handler (High)'Count = 0 =>
      accept Handle (Low)(J : Job) do
        ...
      end;
    end select;
  end loop;
end Worker;
```

Asynchronous abort

- Abort code asynchronously after a timeout or on a signal
- Use **delay** or **delay until** for timeout
- Use entry of protected object for signal

```
select
   delay 5.0;
   ...  -- Do this when aborted
then abort
   ...  -- Abort this code after 5 seconds
end select;

select
   Controller.Wait_Termination;
   ...  -- Do this when aborted
then abort
   ...  -- Abort when entry above is open
end select;
```

Synchronized interfaces

- Ada 2012 added support for synchronized interfaces
  - Task implementation **task interface**
  - Protected object implementation **protected interface**
  - Any implementation **synchronized interface**
- Allows abstraction of tasks and protected objects
- Calls map to entries for tasks, using task rendezvous
- Calls map to entries, procedures and functions for protected objects

Example

```
type SI is synchronized interface;

procedure Handle (This : in out SI; J : in Job) is abstract;

type PI is protected interface and SI;

type TI is task interface and SI;

task type T_Worker is new TI with
  overriding entry Handle (J : in Job);
end T_Worker;

protected type P_Worker is new PI with
  overriding procedure Handle (J : in Job);
end T_Worker;
```

Scheduling

- Several real-time scheduling policies are supported:
  - FIFO within fixed priorities
  - Round-robin within fixed priorities
  - Earliest Deadline First (EDF) within priority range
- Priorities for tasks and interrupts defined in package `System`
- Ceiling priority inheritance protocol for protected objects
- Dynamic priorities for tasks and protected objects
- Asynchronous task control to hold and resume tasks
- Multiprocessor systems support with CPU dispatching domains

Example

```
task type Fixed_Worker (P : Priority) is
   pragma Priority (P);
end Fixed_Worker;

task type EDF_Worker is
   pragma Priority (Some_Priority_In_EDF_Range);
end EDF_Worker;

task body EDF_Worker is
  Next : Time := Clock;
begin
  loop
    Delay_Until_And_Set_Deadline (Next, Milliseconds (10);
    ...
    Next := Next + Milliseconds (100);
  end loop;
end EDF_Worker;
```

Execution time control

- It is possible to monitor the execution time of task and interrupts
  - Clock for tasks and interrupt ID
  - Clock for all interrupt execution
  - Timer for monitoring single task CPU time
  - Group_Budget for monitoring dynamic set of tasks on single CPU
- These features can be used for execution time control of tasks
- Typically pattern is the deferrable server:
  - Replenish budget periodically.
  - Reduce priority of tasks to background when budget is exhausted.
- A group of sporadic tasks can be modeled as one periodic task.

The Ravenscar profile

- The full Ada concurrent constructs have been considered non-deterministic and unsuited for high-integrity applications
- Historically the cyclic-executive has been preferred
- The Ravenscar profile defines a restricted sub-set of the concurrent constructs that are:
  - Deterministic and analyzable
  - Bounded in memory requirements
  - Sufficient for most real-time applications
- The profile also allows for efficient run-time environments by removing features requiring extensive run-time support

Some Ravenscar restrictions

- Tasks and protected objects are only allowed declared statically on library level and tasks may not terminate
- No task entries, tasks communicate only through protected objects and suspension objects
- Protected objects may have at most one entry with a simple Boolean guard and a queue length of one, no requeue
- No dynamic change of task priority with the exception of changes caused by ceiling locking
- No select and asynchronous control

Formal verification with SPARK

- SPARK 2014 is a restricted sub-set of Ada 2012:
  - Heavy use of contract aspects from Ada 2012
  - Additional pragmas for helping proving tools
  - No access types or recursion!
- With SPARK developers can formally verify:
  - Information flow – no uninitalized variables
  - Freedom of run-time errors
  - Functional correctness
  - Security and safety policies
- Easy to get first benefits, full verification requires more...
- Used for high integrity systems such as aviation and security

Summary

- Ada is a programming language most used in safety-critical domains
  - Strong typed and many compiler checks
  - Large systems with packages and abstraction
  - Built-in concurrency and real-time support
- Mature language that has been ISO standard since early 80's
- Latest revision is Ada 2012 with update in 2015
- Excellent tools for a wide range of embedded platforms
- SPARK is a limited sub-set of Ada for formal verification

Thank you!