Langages systèmes 5 - Advanced topics

Raphael Amiard - Adacore
amiard@adacore.com

AdaCore

Ada & embedded programming

Restrictions

```ada
pragma Restrictions (
    No_Implicit_Heap_Allocations,
    No_Exceptions,
    No_Local_Allocators,
    No_Recursion,
    No_Reentrancy,
    No_Unchecked_Conversion,
    No_Unchecked_Deallocation,
    No_Secondary_Stack,
    Static_Priorities
    ...
)
```

- Allow you to select a subset of the language that is amenable to your style of embedded programming
- Standard, and implemented by the compiler rather than by a static analyzer: guaranteed results.

Simple task

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
   task T;

   task body T is
   begin
      Put_Line ("In task T");
   end;
begin
   Put_Line ("In main");
end Main;
```

Simple synchronization

```
procedure P is
   task T;
   task body T is
   begin
      for I in 1 .. 10 loop
         Put_Line ("hello");
      end loop;
   end;
begin
   null;
   --  Will wait here until all tasks have terminated
end;
```

Simple synchronization

```
procedure P is
   task T;
   task body T is
   begin
      for I in 1 .. 10 loop
         Put_Line (“hello”);
      end loop;
   end;
begin
   null;
   --  Will wait here until all tasks have terminated
end;
```

Ravenscar

```
pragma Profile (Ravenscar);
```

is equivalent to

```
pragma Task_Dispatching_Policy (FIFO_Within_Priorities);
pragma Locking_Policy (Ceiling_Locking);
pragma Detect_Blocking;
pragma Restrictions (No_Abort_Statements,
                     No_Calendar,
                     No_Dynamic_Attachment,
                     No_Dynamic_Priorities,
                     No_Implicit_Heap_Allocations,
                     No_Local_Protected_Objects,
                     No_Local_Timing_Events,
                     No_Protected_Type_Allocators,
                     No_Relative_Delay,
                     No_Requeue_Statements,
                     No_Select_Statements,
                     No_Specific_Termination_Handlers,
                     No_Task_Allocators,
                     No_Task_Hierarchy, ...);
```

Ravenscar: Tasking

- All tasks are library level
- None of the tasks is terminating
- Fixed number of tasks
- Deterministic scheduling and locking policies (no preemption, monotonic scheduling, ...)

Querying address

```
with System; use System;

package P is
   V : Integer;
   V_Addr : Address := V'Address;
   --                  ^ Address of V
end P;
```

Querying alignment

```
package P is
   V : Integer;
   V_Alignment : Integer := V'Alignment;
   --                       ^ Alignment of V
end P;
```

AdaCore

Querying size

```
package P is
   V : Integer;

   Typ_Size : Integer := Integer'Size;
   --                    ^ Minimum size for an Integer, in
   --                      bits (== 32)

   Typ_Size : Integer := Natural'Size;
   --                    ^ Minimum size for a Natural, in
   --                      bits (== 31)

end P;
```

Querying size

```
package P is
   Instance_Size : Integer := V'Size;
   --                         ^ Actual size, in bits

   Typ_Size : Integer := Integer'Max_Size_In_Storage_Elements;
   --                    ^ Max size, in storage elements,
   --                      used to store an Int
   --                      (storage element -> byte)
end P;
```

AdaCore

Specifying address

```
with System; use System;

package P is
   V : Integer;

   for V'Address
   use System.Storage_Elements.To_Address (16#fff_0000#);
   --                                     ^ Must be correctly aligned
end P;
```

AdaCore

Specifying address

```
with System; use System;

package P is
   V : Integer
   with Address => System'To_Address (16#fff_0000#);
   --                     ^ GNAT specific attribute

   pragma Import (Ada, V);
   -- Prevents initialization
end P;
```

Specifying address

```ada
procedure Pouet is
   A : array (1 .. 32) of Integer;

   B : array (1 .. 32 * 4) of Character
   with Address => A'Address;
   --  B is now an overlay for A, except you manipulate
   --  memory in bytes.

   type Rec is record
      A, B : Integer;
   end Rec;

   Inst : Rec;

   C : Integer
   with Address => Inst'Address;
begin
   null;
end Pouet;
```

Specifying size

```
with System; use System;

package P is
   V : Natural
   with Size => 32;
   --           ^ Must be large enough.
   --           ^ Compiler can choose bigger size.
end P;
```

AdaCore

Specifying alignment

```
with System; use System;

package P is
   V : Integer;
   with Alignment => 1;
   --                 ^ Address must be a multiple of this.
   --                   So compiler can over align.
end P;
```

Packing arrays

```
procedure BV is
   type Bit_Vector is array (0 .. 31) of Boolean;
   pragma Pack (Bit_Vector);

   B : Bit_Vector;
begin
   Put_Line (Integer'Image (B'Size));
   -- Prints 32
end;
```

AdaCore

Packing records

```
procedure Packed_Rec is
   type My_Rec is record
      A : Boolean;
      C : Natural;
   end record
   with Pack;

   R : My_Rec;
begin
   Put_Line (Integer'Image (R'Size));
   -- Prints 32
end Packed_Rec;
```

AdaCore

Specifying record layout

```
type Register is range 0 .. 15;
   with Size => 4;
--  Size on type only affects components

type Opcode is (Load, Inc, Dec, ..., Mov);
   with Size => 8;

type RR_370_Instruction is record
   Code : Opcode;
   R1   : Register;
   R2   : Register;
end record;

for RR_370_Instruction use record
   Code at 0 range 0 .. 7;
   R1 at 1 range 0 .. 3;
   R2 at 1 range 4 .. 7;
end record;
```

Specifying how enums are mapped to numbers

```
type My_Boolean is new Boolean;
for My_Boolean use (False => 3, True => 6);
```

Bit to bit conversion

```
with Ada.Unchecked_Conversion;

procedure Unconv is
   subtype Str4 is String (1 .. 4);
   function To_Str4 is new Ada.Unchecked_Conversion (Integer, Str4);

   V : Integer;
   S : Str4;
   S := To_Str4 (V)
begin
   null;
end Unconv;
```

AdaCore

Pragma Volatile

```ada
type Video_Buffer is array (Natural range <>) of RGB_Value;
pragma Volatile (Video_Buffer);
```

- Value might change at any time
- Compiler cannot optimize reads/writes, has to compile *exactly* each read and write the programmer means.

Case study: Register overlays (1)

```
package Registers is

   type Bit    is mod 2 ** 1
     with Size => 1;
   type UInt5  is mod 2 ** 5
     with Size => 5;
   type UInt10 is mod 2 ** 10
     with Size => 10;

   subtype USB_Clock_Enable is Bit;
```

Case study: Register overlays (2)

```ada
type PMC_SCER_Register is record --  System Clock Enable Register
   Reserved_0_4   : UInt5            := 16#0#; -- Reserved bits
   USBCLK         : USB_Clock_Enable := 16#0#; -- Write-only. Enable USB FS Clock
   Reserved_6_15  : UInt10           := 16#0#; -- Reserved bits
end record
  with
    Volatile,
    Size      => 16,
    Bit_Order => System.Low_Order_First;

for PMC_SCER_Register use record
   Reserved_0_4   at 0 range 0 .. 4;
   USBCLK         at 0 range 5 .. 5;
   Reserved_6_15  at 0 range 6 .. 15;
end record;
```

Case study: Register overlays (3)

```
   type PMC_Peripheral is record --  Power Management Controller
      PMC_SCER       : aliased PMC_SCER_Register; -- System Clock Enable Register
      PMC_SCDR       : aliased PMC_SCER_Register; -- System Clock Disable Register
   end record
     with Volatile;

   for PMC_Peripheral use record
      PMC_SCER       at 16#0# range 0 .. 15; --  16-bit register at byte 0
      PMC_SCDR       at 16#2# range 0 .. 15; --  16-bit register at byte 2
   end record;

   --  Power Management Controller
   PMC_Periph : aliased PMC_Peripheral
     with Import, Address => System'To_Address (16#400E0600#);

end Registers;
```

Rust macros

Macro example

```
#[macro_export]
macro_rules! vec {
    ( $( $x:expr ),* ) => {
        {
            let mut temp_vec = Vec::new();
            $(
                temp_vec.push($x);
            )*
            temp_vec
        }
    };
}
```

AdaCore

Macros

- Macros allow you to extend your language's syntax and semantics, by extending what it's able to do at compile time.
- Bad macros work on text (like C-like macros)
- Good macros work on structured input
  - In lisp, it worked on lists
  - In Rust, it works on ASTs, token trees, or token streams.

When to use macros

- Never
- Never

But seriously though

- Macros are used to:
  - Abstract common repetitive programming patterns
  - Embed domain specific languages
  - Provide lazy evaluation
- Generally: Macros are a last resort. Anything that you can solve another way shouldn't be fixed with macros.

The rust parsing pipeline

First, tokenization: Turn string into tokens.

From there, Rust can either:

1. Produce a syntax tree from tokens
2. Produce a token tree that your macro can match upon

Declarative macros

- Like the `vec` one.
- You use a form of pattern-matching on the arguments
- Used to provide function-like macros

Declarative macros (2)

```rust
macro_rules! ok_or_return {
    ($e:expr, $err:expr) => {
        {
            match $e {
                Ok(value) => value,
                Err(_) => return Err($err)
            }
        }
    }
}

fn main() -> Result<(), &'static str> {
    let mut line = String::new();
    ok_or_return!(std::io::stdin().read_line(&mut line), // including '\n'
                  "Cannot read line");
    let a = ok_or_return!(line.trim().parse::<i32>(), "Cannot parse string");
    Ok(())
}
```

Declarative macros - Variadic arguments

```
macro_rules! vec_strs {
    (
        // Start a repetition:
        $(
            $element:expr // Each repeat must contain an expression...
        )
        , // ...separated by commas...
        * // ...zero or more times.
    ) => {
        // Enclose the expansion in a block so that we can use
        // multiple statements.
        {
            let mut v = Vec::new();
            // Start a repetition:
            $(
                // Each repeat will contain the following statement, with
                // $element replaced with the corresponding expression.
                v.push(format!("{}", $element));
            )*
            v
        }
    };
}
```

Hygiene

```c
#define INCI(i) do { int a=0; ++i; } while (0)
int main(void)
{
    int a = 4, b = 8;
    INCI(a);
    INCI(b);
    printf("a is now %d, b is now %d\n", a, b);
    return 0;
}
```

Hygiene and Rust

```rust
macro_rules! using_a {
    ($e:expr) => {
        {
            let a = 42;
            $e
        }
    }
}

let four = using_a!(a / 10); // Won't work
```

AdaCore

Procedural macros

```rust
use proc_macro::TokenStream;

#[proc_macro]
pub fn tlborm_fn_macro(input: TokenStream) -> TokenStream {
    input
}
```