Proving the Correctness of GNAT Light Runtime Library

Yannick Moy - FOSDEM 2022

AdaCore Capgemini engineering

PARTNERSHIP

GNAT Light Runtime Library

Previously known as zfp / ravenscar / cert runtimes

Targeted at **embedded platforms**: 77 platforms both baremetal (ARM, Leon, PowerPC, RISC-V, x86, x86_64) and with OS (PikeOS, VxWorks)

Ready for **certification**: avionics (DO-178), space (ECSS-E-ST40C), railway (EN 50128), automotive (ISO-26262)

Build scripts available at https://github.com/AdaCore/bb-runtimes
Sources in GCC repository

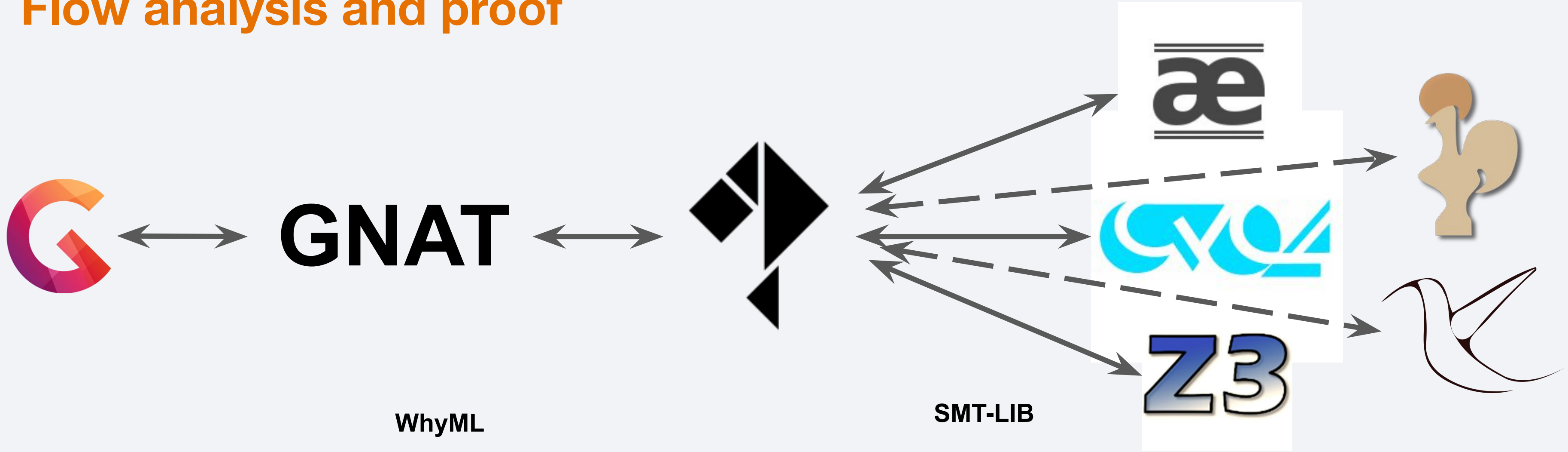
Tour of the GNAT Light Runtime Library

Version for x86_64 has 182 spec files (.ads) split as follows:

- Ada standard library (26 a-*.ads, 4 i-*.ads, a few others)
    - Character and string handling
    - Numerics library
    - Assertions, exceptions (but no propagation)
    - Interface with C
- GNAT user library (4 g-*.ads), mostly IO
- GNAT runtime library (140 s-*.ads)
    - Support for attributes ‘Image, ‘Value, ‘Width and attributes of floats
    - Support for arithmetic operations (fixed-points, floats, exponentiation) and numerics
    - Support for tasking

SPARK - Formal Verification Tool

Flow analysis and proof



SPARK

```
A(1) := 42;
```

WhyML

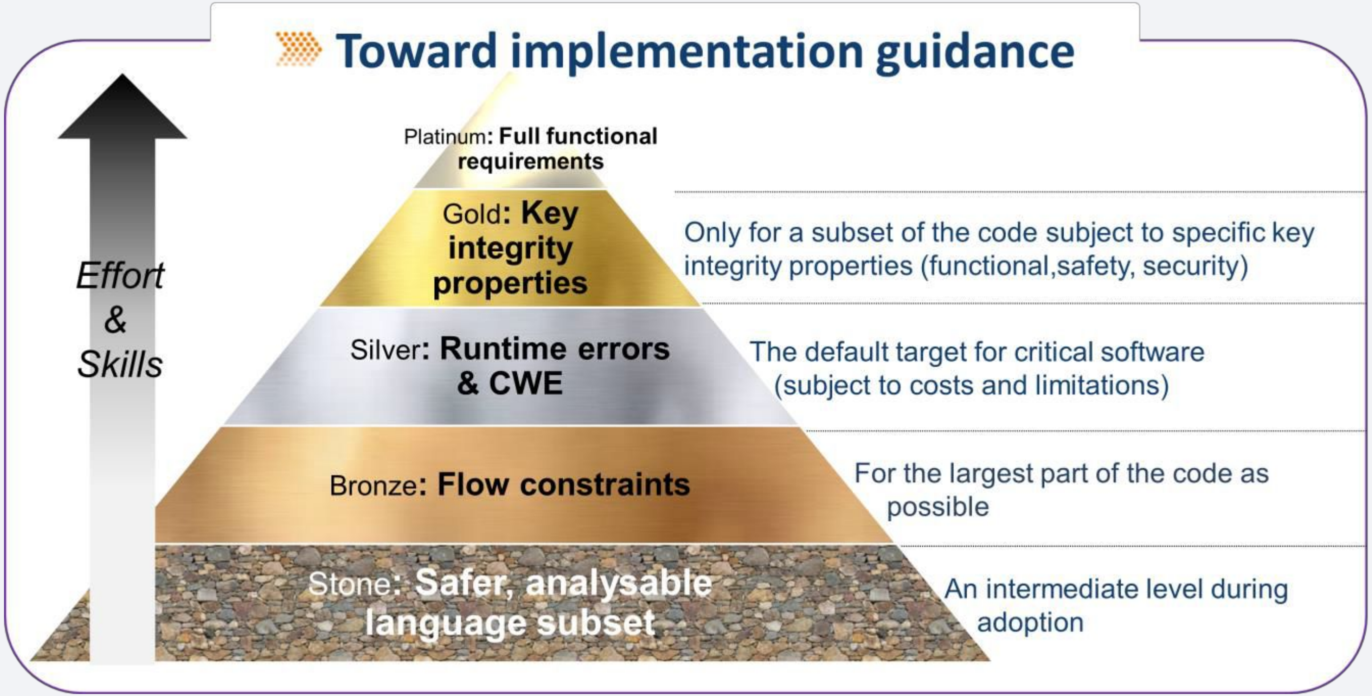
```
a.map__content <-
  set
    (a.map__content)
    (let temp = 1 : int in
       assert { temp ... };
       temp)
    (42 : value)
```

SMT-LIB

```
(assert
  (not
  (=> (dynamic_property 0 1000000
         (to_rep a__first) (to_rep a__last))
  (=> (and (= (to_rep a__first) 1)
           (<= 0 (to_rep a__last)))
  (<= (to_rep a__first) 1)))))
(check-sat)
```

SPARK - Software Assurance Levels

A pragmatic view of costs & benefits



AdaCore

SPARK - Software Assurance Levels

A pragmatic view of costs & benefits

Motivating Example

2012 - Adding Support for Bigint in GNAT

Motivated by feature of SPARK to allow intermediate computations without possible overflows

Implemented by late great developer Robert Dewar, using Knuth’s algorithm D for multi-precision division (TAOCP Vol 2, 2nd Edition - 1981, section 4.3.1)

Reviewer> possible overflow in ((u (j) & u (j + 1)) - DD (qhat) * DD (v1)) * b
Robert   > show me an actual issue!
Reviewer> ...
Reviewer> try ((((2^32 - 2) * 2^32) + 2^32 - 2) * 2^32) / ((2^32 - 2) * 2^32 + 2^32 - 1)
Robert   > true result is 4_294_967_295 but Knuth gives 2_147_483_648 !?!

Even the Best Can Get Details Wrong…

Especially when it comes to overflows

Bug already fixed in 1995, in errata of Vol 2, 2nd Edition:

Page 258 first three lines of step D3 ________________ 28 Sep 1995

If $u_j = v_1$, . . . latter test determines $\rightsquigarrow$ Set $\hat{q} \leftarrow \lfloor (u_j b + u_{j+1})/v_1 \rfloor$ and $\hat{r} \leftarrow (u_j b + u_{j+1}) \bmod v_1$. Now test if $\hat{q} = b$ or $v_2 \hat{q} > b\hat{r} + u_{j+2}$; if so, decrease $\hat{q}$ by 1, increase $\hat{r}$ by v_1, and repeat this test if $\hat{r} < b$. [The test on v_2 determines

Even the Best Can Get Details Wrong…

Especially when it comes to overflows

Bug already fixed in 1995, in errata of Vol 2, 2nd Edition:

Page 258 first three lines of step D3 ________ 28 Sep 1995

If $u_j = v_1$, $\ldots$ latter test determines $\leadsto$ Set $\hat{q} \leftarrow \lfloor (u_j b + u_{j+1})/v_1 \rfloor$ and $\hat{r} \leftarrow (u_j b + u_{j+1}) \bmod v_1$. Now test if $\hat{q} = b$ or $v_2\hat{q} > b\hat{r} + u_{j+2}$; if so, decrease $\hat{q}$ by 1, increase $\hat{r}$ by v_1, and repeat this test if $\hat{r} < b$. [The test on v_2 determines

Even the Best Can Get Details Wrong…

Especially when it comes to overflows

Bug already fixed in 1995, in errata of Vol 2, 2nd Edition:

Page 258 first three lines of step D3 ———— 28 Sep 1995

If $u_j = v_1$, … latter test determines $\rightsquigarrow$ Set $\hat{q} \leftarrow \lfloor (u_j b + u_{j+1})/v_1 \rfloor$ and $\hat{r} \leftarrow (u_j b + u_{j+1}) \bmod v_1$. Now test if $\hat{q} = b$ or $v_2 \hat{q} > b\hat{r} + u_{j+2}$; if so, decrease $\hat{q}$ by 1, increase $\hat{r}$ by v_1, and repeat this test if $\hat{r} < b$. [The test on v_2 determines

… and further fixed in 2005, in errata of Vol 2, 3rd Edition:

▸**Page 272** line 2 of step D3 ———— 03 Feb 2005

test if $\hat{q} = b \rightsquigarrow$ test if $\hat{q} \geq b$

Algorithms Get Implemented Everywhere…

Could the same bug occur elsewhere?

Algorithm D also used in other units:

- in uintp.adb for arbitrary-precision computation at compile time
- in s-arit64.adb for support of fixed-point arithmetic

But no two implementations are alike…

Fix propagated to uintp.adb despite absence of clear bug

No bug could be identified in s-arit64.adb which uses a different comparison

2019 - Runtime Certification for Space

A close call on critical software

External reviewer of certification material suggests to increase comment frequency on implementation of algorithm D for Scaled_Divide in s-arit64.adb

New internal review detects 2 possible (silent) overflows in Double_Divide and a missing exception in Scaled_Divide 

Colleague > I challenge the SPARK team to prove that unit!
SPARK team> Let's see what we can do.
….1 week of work later…
SPARK team> We got all algorithms proved except Scaled_Divide, worth doing?
…moving on…

Using free icon from https://icons8.com

AdaCore

2021 - Summer Internship

Intern Pierre-Alexandre Bazin updates previous proofs and proves Scaled_Divide (now in generic unit s-aridou.adb)

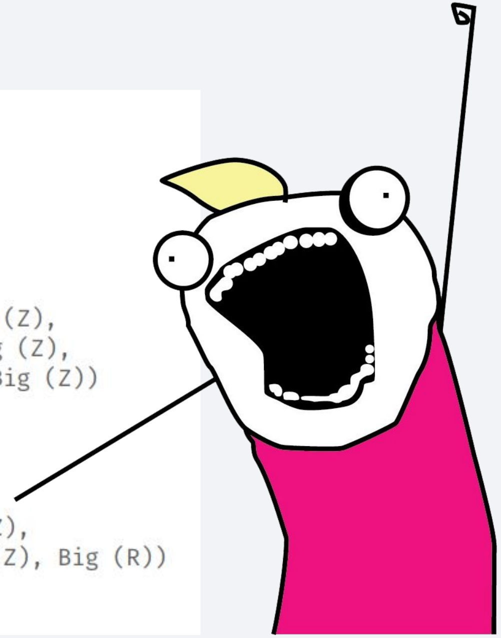
```
procedure Scaled_Divide
  (X, Y, Z : Double_Int;
   Q, R    : out Double_Int;
   Round   : Boolean)
with
  Pre  => Z /= 0
    and then In_Double_Int_Range
      (if Round then Round_Quotient (Big (X) * Big (Y), Big (Z),
                                     Big (X) * Big (Y) / Big (Z),
                                     Big (X) * Big (Y) rem Big (Z))
       else Big (X) * Big (Y) / Big (Z)),
  Post => Big (R) = Big (X) * Big (Y) rem Big (Z)
    and then
      (if Round then
         Big (Q) = Round_Quotient (Big (X) * Big (Y), Big (Z),
                                   Big (X) * Big (Y) / Big (Z), Big (R))
       else
         Big (Q) = Big (X) * Big (Y) / Big (Z));
```

AdaCore

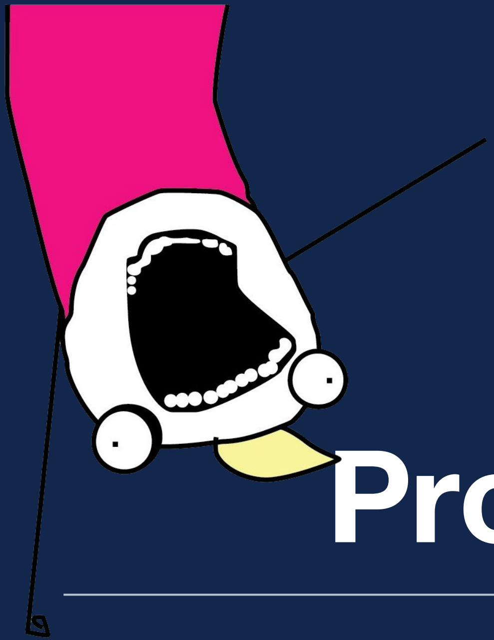
2021 - Summer Internship

Intern Pierre-Alexandre Bazin updates previous proofs and proves Scaled_Divide (now in generic unit s-aridou.adb)

```ada
procedure Scaled_Divide
  (X, Y, Z : Double_Int;
   Q, R    : out Double_Int;
   Round   : Boolean)
with
  Pre  => Z /= 0
    and then In_Double_Int_Range
      (if Round then Round_Quotient (Big (X) * Big (Y), Big (Z),
                                     Big (X) * Big (Y) / Big (Z),
                                     Big (X) * Big (Y) rem Big (Z))
       else Big (X) * Big (Y) / Big (Z)),
  Post => Big (R) = Big (X) * Big (Y) rem Big (Z)
    and then
      (if Round then
         Big (Q) = Round_Quotient (Big (X) * Big (Y), Big (Z),
                                   Big (X) * Big (Y) / Big (Z), Big (R))
       else
         Big (Q) = Big (X) * Big (Y) / Big (Z));
```



AdaCore



Prove All The Things

But… the Runtime is not in SPARK

Untyped handling of memory for secondary stack, array comparison, OO support (tags), binding to C strings…

Use of type `Address` in Ada and unchecked conversion to pointers

→ not supported by ownership system in SPARK

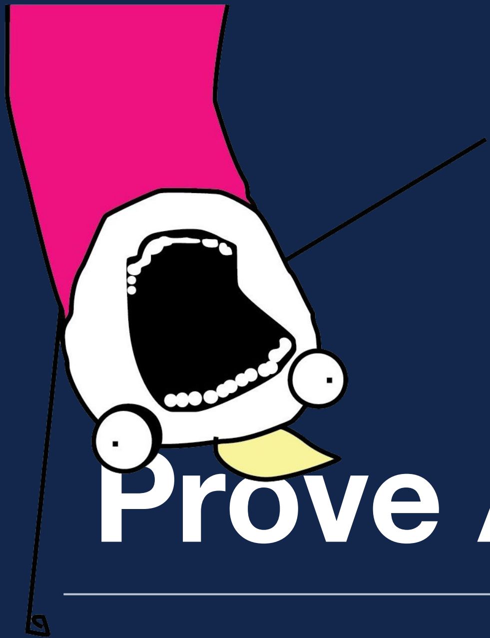
And… not Everything is Provable

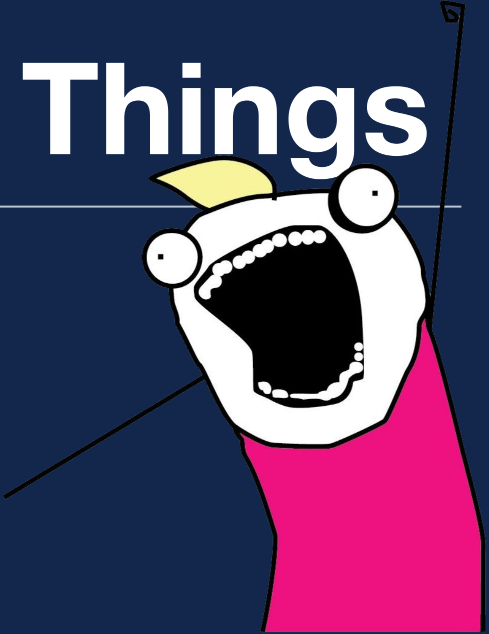
Low-level support of floating-point operations for language attributes, numerics (trigonometry), double arithmetic from floats…

Depends on bit-representation of floats (overlays, NaN/Inf) and complex floating-point reasoning

→ not supported by model of floats and provers in SPARK

Prove All The SPARK Things





Proving the Interface to C

`Interfaces.C`

Need to define ghost function `C_Length_Ghost`:

- expresses in the spec the value of C function `strlen`
- can be used in contracts
- is implemented and proved itself

Proof makes heavy (but simple) use of advanced SPARK features:

- loop invariants to summarize state at current loop iteration
- relaxed initialization of uninitialized local array variables

Proving Fixed-Point Support

`System.Arith_32`, `System.Arith_64`, `System.Arith_Double`: generic code only proved in the context of an instantiation

Comments in code translated into SPARK contracts

Ex: `Add_With_Ovflo_Check` in `System.Arith_64`

```
function In_Int64_Range (Arg : Big_Integer) return Boolean is
  (Ada.Numerics.Big_Numbers.Big_Integers_Ghost.In_Range
    (Arg, Big (Int64'First), Big (Int64'Last)))
with Ghost;

function Add_With_Ovflo_Check64 (X, Y : Int64) return Int64
with
  Pre  => In_Int64_Range (Big (X) + Big (Y)),
  Post => Add_With_Ovflo_Check64'Result = X + Y;
--  Raises Constraint_Error if sum of operands overflows 64 bits,
--  otherwise returns the 64-bit signed integer sum.
```

AdaCore

Proving Character and String Handling

`Ada.Characters.Handling`

`Ada.Strings.Bounded, Ada.Strings.Fixed, Ada.Strings.Maps`: these plus supporting units in GNAT

Ada RM description translated into SPARK contracts

Ex: `Is_Control` in `Ada.Characters.Handling` is *“True if Item is a control character. A control character is a character whose position is in one of the ranges 0..31 or 127..159.”*

```
function Is_Control              (Item : Character) return Boolean
with
  Post => Is_Control'Result =
    (Character'Pos (Item) in 0 .. 31 | 127 .. 159);
--  True if Item is a control character. A control character is a character
--  whose position is in one of the ranges 0..31 or 127..159.
```

AdaCore

Proving Exponentiation Support

`System.Expont`, `System.Exponn`, `System.Exponu`, `System.Expmod` and all instantiations

Binary modular:

```
function System.Exponu (Left : Int; Right : Natural) return Int
with
  SPARK_Mode,
  Post => System.Exponu'Result = Left ** Right;
```

Signed:

```
function Expon (Left : Int; Right : Natural) return Int
with
  Pre  => In_Int_Range (Big (Left) ** Right),
  Post => Expon'Result = Left ** Right;
```

Non-binary modular:

```
function Exp_Modular
  (Left    : Unsigned;
   Modulus : Unsigned;
   Right   : Natural) return Unsigned
with
  Pre  => Modulus /= 0 and then Modulus not in Power_Of_2,
  Post => Big (Exp_Modular'Result) = Big (Left) ** Right mod Big (Modulus);
```

Proving Support for ‘Image and ‘Value

`System.Img_Bool, System.Val_Bool, System.Value_U, System.Value_I` and all instantiations

Specification that `Image` and `Value` are reverse functions:

- precise postcondition for `Value`
- so that postcondition for Image can state `Value (Image’Result (V)) = V`

```
procedure Image_Boolean
  (V : Boolean;
   S : in out String;
   P : out Natural)
with
  Pre  ⇒ S'First = 1
    and then (if V then S'Length ≥ 4 else S'Length ≥ 5),
  Post ⇒ (if V then P = 4 else P = 5)
    and then System.Val_Bool.Is_Boolean_Image_Ghost (S (1 .. P))
    and then System.Val_Bool.Value_Boolean (S (1 .. P)) = V;
```

AdaCore

Current Status

Issues Detected and Fixed

Possible overflow / range check failures

Ex on support of ‘Value:

```
procedure Test_Value is
   S : String(Natural'Last .. Natural'Last) := " ";
   B : Boolean;
begin
   B := Boolean'Value (S);
end Test_Value;
```

with GNAT Community 2020: `segmentation fault (core dumped)`

with GNAT Community 2021: `raised CONSTRAINT_ERROR : s-valuti.adb:79 index check failed`

with current GNAT FSF: `raised CONSTRAINT_ERROR : bad input for 'Value: " "`

Partial Proof of GNAT Light Runtime Library

35 units functionally specified and proved (out of 180)

Daily proof takes 1.5h on 36 cores Linux server (3 configs: x86_64-linux, aarch64-vx7r2cert-linux64, aarch64-elf-linux64)

Many specifications added: 393 preconditions, 508 postconditions

Proof requires addition of ghost code: 146 loop invariants, 381 assertions, 270 ghost entities (of which 152 lemmas)

Can this effort benefit future certifications of the runtime?

Can we go beyond what SPARK currently supports?

Thank You

www.adacore.com   @AdaCoreCompany

Yannick Moy   moy@adacore.com

AdaCore