Introduction to SPARK 2014
How to Develop Ultra-Low Defect Software

Do setup now: www.rcs.ei.tum.de/spark2014-intro

Martin Becker
Technical University of Munich
Real-Time Computer Systems

September 18th, 2017
Forum on Specification & Design Languages
Verona, Italy
Introduction

⏰ 0min
Motivation for SPARK 2014

NASA’s CubeSat Launch Initiative, launch “ELaNa IV”

- 11 institutions (one of them NASA Research Center) built mini satellites
- launch to 500km LEO in 2013:
Motivation for SPARK 2014

NASA’s CubeSat Launch Initiative, launch “ELaNa IV”

- 11 institutions (one of them NASA Research Center) built mini satellites
- launch to 500km LEO in 2013:
  - 8 satellites went missing
Motivation for SPARK 2014

NASA’s CubeSat Launch Initiative, launch “ELaNa IV”

- 11 institutions (one of them NASA Research Center) built mini satellites
- launch to 500km LEO in 2013:
  - 8 satellites went missing
  - 2 worked for < 1 week
Motivation for SPARK 2014

NASA’s CubeSat Launch Initiative, launch “ELaNa IV”

- 11 institutions (one of them NASA Research Center) built mini satellites
- launch to 500km LEO in 2013:
  - 8 satellites went missing
  - 2 worked for < 1 week
  - 1 worked for 4 months
Motivation for SPARK 2014

NASA’s CubeSat Launch Initiative, launch “ELaNa IV”

- 11 institutions (one of them NASA Research Center) built mini satellites
- launch to 500km LEO in 2013:
  - 8 satellites went missing
  - 2 worked for < 1 week
  - 1 worked for 4 months
  - 1 worked until re-entry after 2 years and 293,000,000 miles
Motivation for SPARK 2014

NASA’s CubeSat Launch Initiative, launch “ELaNa IV”

■ 11 institutions (one of them NASA Research Center) built mini satellites
■ launch to 500km LEO in 2013:
  ■ 8 satellites went missing
  ■ 2 worked for < 1 week
  ■ 1 worked for 4 months
  ■ 1 worked until re-entry after 2 years and 293,000,000 miles
    ■ satellite of Vermont Technical College
Motivation for SPARK 2014

NASA’s CubeSat Launch Initiative, launch “ELaNa IV”

- 11 institutions (one of them NASA Research Center) built mini satellites
- launch to 500km LEO in 2013:
  - 8 satellites went missing
  - 2 worked for < 1 week
  - 1 worked for 4 months
  - 1 worked until re-entry after 2 years and 293,000,000 miles
    - satellite of Vermont Technical College
    - powered by SPARK’05
The SPARK 2014 Language


- conceived with verification in mind
- imperative language with good tool and platform support
  - object-oriented, high re-use
- designed for embedded and real-time systems
  - run-time checks, strongly typed, inherent support for parallelism
The SPARK 2014 Language

- conceived with verification in mind
- imperative language with good tool and platform support
  - object-oriented, high re-use
- designed for embedded and real-time systems
  - run-time checks, strongly typed, inherent support for parallelism
- **principal of minimal surprise**: explicit, easily readable for humans, structural analogies, not case sensitive:

  ```
  Vmax : constant Speed := 12_512.0 * Meter/Second;
  ```
The SPARK 2014 Language


- conceived with verification in mind
- imperative language with good tool and platform support
  - object-oriented, high re-use
- designed for embedded and real-time systems
  - run-time checks, strongly typed, inherent support for parallelism
- principal of minimal surprise: explicit, easily readable for humans, structural analogies, not case sensitive:
  \[
  \texttt{Vmax} : \texttt{constant Speed} := 12.512.0 \ast \texttt{Meter/Second};
  \]
- difference to C(++): Getting it to compile is harder, but much less bugs after (longer development time vs. more maintenance)
The SPARK 2014 Language


- conceived with verification in mind
- imperative language with good tool and platform support
  - object-oriented, high re-use
- designed for embedded and real-time systems
  - run-time checks, strongly typed, inherent support for parallelism
- **principal of minimal surprise**: explicit, easily readable for humans, structural analogies, not case sensitive:
  
  ```
  Vmax : constant Speed := 12_512.0 * Meter/Second;
  ```

- **difference to C(++)**: Getting it to compile is harder, but much less bugs after (longer development time vs. more maintenance)

- rated by NIST as approach with “dramatic future impact” (reducing vulnerabilities by two orders of magnitude)
Pedigree of SPARK 2014
Famous users of Ada and SPARK

- Air Traffic Management: AU, CA, FR, DE, NZ, UK, USA, ...
- Civil Aviation: Boeing 777 (99.9%!), A330, Tu-204, ...
- Railway: TGV, ETCS, ...
- Rockets: Ariane, Delta, ...
- Satellites: INMARSAT, ...
- Banking: Reuters, ...
- Medical: JEOL Nuclear MRI, LifeFlow VAD, ...
- Military: Eurofighter combat aircraft, ...

...
The next 82 minutes...

1. Hands-on Part I
2. Basic Features
3. Outsmart programmers from IBM and Bell Labs
4. Hands-on Part II
5. Proof
6. Hands-on Part III
7. Advanced Topics
8. Wrap-Up
Hands-on Part I

⏱ 8min
Setting up “Hello World” (10min)

Step by step:
- GPS IDE
- functions, procedures
- packages
- variables
- Put(Line (with, use)
- if-then

Final result: http://tinyurl.com/yabb8ktm
Basic Features

⏰ 18min
Data Types in Ada/SPARK 2014

**Strong Type System:** “I cannot even assign this integer to that other integer...”
Data Types in Ada/SPARK 2014

**Strong Type System:** “I cannot even assign this integer to that other integer...”

Weakly typed means (C++, C, Java, ...):
- typedefs are not real types
- implicit conversions from int to float
- weak overflow semantics/undefined behavior
- missing types (fixed-point)
- two objects of different types can be mixed, although they shouldn’t: one can add up variables representing *speed* and *time*

Ada/SPARK is the opposite.
create a new type based on an existing one. It is *incompatible* to the existing one, but inherits the operations (+,-,*,/,...)

```haskell
type Int_1 is new Integer range 1..10; -- declare type
type Int_2 is new Integer range 1..10;
var1 : Int_1; -- create object/instance of type
var2 : Int_2;
var1 := var2; -- ERROR: type mismatch
```
User-Defined Types

- create a new type based on an existing one. It is \textit{incompatible} to the existing one, but inherits the operations (+,-,\text{*},/,...)

  \begin{verbatim}
  type Int_1 is new Integer range 1..10; -- declare type
type Int_2 is new Integer range 1..10;
  var1 : Int_1; -- create object/instantiation of type
  var2 : Int_2;
  var1 := var2; -- ERROR: type mismatch
  \end{verbatim}

- a \textit{subtype} is compatible to the base type:

  \begin{verbatim}
  subtype Integer_1 is Integer range 1..10;
  var : Integer;
  var1 : Integer_1;
  var := var1; -- no problem
  \end{verbatim}
User-Defined Types

- create a new type based on an existing one. It is **incompatible** to the existing one, but inherits the operations (+,-,*,/,...)

  ```
  type Int_1 is new Integer range 1..10; -- declare type
  type Int_2 is new Integer range 1..10;
  var1 : Int_1; -- create object/instance of type
  var2 : Int_2;
  var1 := var2; -- ERROR: type mismatch
  ```

- a **subtype** is compatible to the base type:

  ```
  subtype Integer_1 is Integer range 1..10;
  var : Integer;
  var1 : Integer_1;
  var := var1; -- no problem
  ```

- way number 3: create new type without base type (nothing inherited)

  ```
  type My_Currency is digits 6;
  type My_Enum is (HI, LO, Z);
  ```
Attributes

- get/set information about an object
- denoted by a single quote, e.g.:
  - myInteger’First. Attribute “First” returns lowest possible value in this type
  - myInteger’Last. Returns highest possible value in this type
## Attributes

- get/set information about an object
- denoted by a single quote, e.g.:
  - `myInteger'First`. Attribute “First” returns lowest possible value in this type
  - `myInteger'Last`. Returns highest possible value in this type
- attribute “Range” can be used to iterate over arrays:

```vhdl
myArray : array (1..5) of Integer := (2,4,6,8,10);
for i in myArray'Range loop
  -- ...
end loop;
```
Attributes

- get/set information about an object
- denoted by a single quote, e.g.:
  - `myInteger'First`. Attribute “First” returns lowest possible value in this type
  - `myInteger'Last`. Returns highest possible value in this type
- attribute “Range” can be used to iterate over arrays:
  ```
  myArray : array(1..5) of Integer := (2,4,6,8,10);
  for i in myArray'Range loop
      -- ...
  end loop;
  ```
- Attributes “Succ” and “Pred” can be used to walk through discrete types (e.g., enums, where you cannot do “+1”):
  ```
  type My_Weekdays is (Monday, Holiday, Friday);
  w : My_Weekday := Monday;
  w := My_Weekday'Succ (w); -- now we got Holiday
  ```

For more see cheat sheet.
Exceptions in SPARK/Ada

- notify when something is about to go wrong:
  - reading beyond an array
  - assigning a value beyond range of type
  - ...

- many things are checked during run-time
  - DIV/0, overflow, array access, range checks, ...
  - unlike C language (it just continues execution)
  - (we could disable these checks, but ...)
Exceptions in SPARK/Ada

- notify when something is about to go wrong:
  - reading beyond an array
  - assigning a value beyond range of type
  - ...
- many things are checked during run-time
  - DIV/0, overflow, array access, range checks, ...
  - unlike C language (it just continues execution)
  - (we could disable these checks, but ...)
- but SPARK 2014 *forbids* the use of exception handlers
- forces the developer to handle corner cases explicitly
  - unhandled exceptions: program terminated
  - we must write programs that are free of exceptions
Exceptions in Ada (2)

The following causes an exception at line 5:

```ada
procedure main is
  type mydays is new Integer range 1 .. 31
  d : mydays := 31;
begind := d + 1;
  Put_Line("day is:" & mydays'Image(d)); --unreachable
end main;
```

Some exceptions are foreseen by the compiler. Others are not.
Outsmart programmers from IBM and Bell Labs

⏰ 27min
The binary search algorithm

Searching for a number ("key") in a sorted array, return position

Example: searching for key=23

1. L = first element, H = last element
2. Test element in the middle of H - L:
   - If element = key: found, terminate.
   - If element < key: repeat search in right half (L = middle + 1).
   - If element > key: repeat search in left half (H = middle - 1).

Image (modified) from http://www.geeksforgeeks.org/binary-search/
The binary search algorithm

Searching for a number ("key") in a sorted array, return position

Example: searching for key=23

1. L=first element, H=last element
2. test element in the middle of H-L:
   - if element = key: found, terminate.
   - if element < key: repeat search in right half (L=middle+1).
   - if element > key: repeat search in left half (H=middle-1).

image (modified) from http://www.geeksforgeeks.org/binary-search/
Hands-on Part II

ござ 30min
Binary Search in SPARK

Start with this skeleton (binary_search.ads):

```ada
package Binary_Search is
type Arr_T is array (Positive range <>) of Integer;

function Search (Arr : Arr_T; Key : Integer) return Natural;
end Binary_Search;
```

`main.adb`:

```ada
with Binary_Search; use Binary_Search;
with Ada.Text_IO; use Ada.Text_IO;

procedure main is
  arr : constant Arr_T (1..10) := (1, 2, 4, 5, 6, 7, 8, 9, 10, 11);
  idx : Natural;
begin
  -- let's test one case:
  idx := Search (arr, 4);
  Put_Line ("Result is " & Natural’Image(idx));
end main;
```

Write `binary_search.adb` yourself, and test some cases.

Copy and paste from: http://tinyurl.com/yash48gs
One more Test Case

Modify your main.adb as follows:

```ada
Arr (1..Natural’Last) := (Natural’Last => 1, others => 0);
-- ...
idx := Search (arr, 1);
Put_Line ("Result is " & Natural’Image(idx));
```

One of many possible mistakes (low+high overflows).
Solution?
One more Test Case

Modify your main.adb as follows:

```ada
Arr (1..Natural’Last) := (Natural’Last => 1, others => 0);
-- ...
idx := Search (arr, 1);
4 Put_Line ("Result is " & Natural’Image(idx));
```

One of many possible mistakes (low+high overflows).
Solution?

```ada
mid := low + (high - low);
```
Proof

⏰ 47min
Motivation for Proof

So far we are only using Ada, but without exception handling.
Motivation for Proof

So far we are only using Ada, but without exception handling.

Two possible ways forward:

1. use full Ada: handle exceptions, apply testing
2. use SPARK subset: no exception handling, find proof
Launching the verification tool (GNATprove)

1. Please add with `SPARK_Mode` to your search package (see cheat sheet)
2. Click on menu “SPARK” → “Prove All Sources”
3. Select “Multiprocessing”, “Report checks proved” and “Proof level=2”
4. Click on “Execute”

Finds all defects that could lead to exceptions – most likely you will have a few red lines now...

Fixing all these lines would guarantee *Absence of Run-time Error* (AoRTE)
Verification of SPARK 2014

- static analysis considers all possible inputs and interactions
- equivalent to exhaustive testing, but
  - no test harness or fixture necessary
  - usually much faster than extensive testing
  - exhaustive testing not possible in real world
- some properties are over-approximated, but in a safe way:
  - unless verifier finds a proof that bad things are impossible, properties are flagged as failing (red)
  - e.g.: $a+b$ is classified as overflow, if ranges of $a$ and $b$ are unspecified
- sometimes verifier needs a little help (e.g., with loops)
Verification of SPARK 2014

Beyond this tutorial...

“Any sufficiently advanced technology is indistinguishable from magic.”

Arthur C. Clarke
Verification of SPARK 2014

Beyond this tutorial...

“Any sufficiently advanced technology is indistinguishable from magic.”

Arthur C. Clarke
Another Type of Error

Program works perfectly, for all possible inputs
- no crash, no silent overflows, ...
- already better than C++, Java, Rust et al.
- but . . .
Another Type of Error

Program works perfectly, for all possible inputs
- no crash, no silent overflows, ...
- already better than C++, Java, Rust et al.
- but it could still do the wrong thing
  - Not searching at all
  - search only half of the array
  - ...
- we need the option to specify intended behavior
- and then also prove it!
Contracts

- formal agreement between the implementer and the user of a program unit (package or subprogram)
- assigns responsibilities
- a way to organize and document your code
- not a new idea (Floyd, Hoare, Dijkstra, Meyer)
Pre- and Postconditions

- are Boolean expressions
- $\text{Pre}$ is an obligation of the caller
- $\text{Post}$ is an obligation of the callee
- are verified, just like any other potential exception
Precondition Contracts

- conditions that must hold true when the subprogram is called

  -- example:
  ```
  procedure Increment (X : in out Integer) 
  with Pre => X < Integer'Last; 
  ```

- responsible: caller (must obey these conditions)
- verifier tries to prove every subprogram call
Precondition Contracts

- conditions that must hold true when the subprogram is called

```
procedure Increment (X : in out Integer)
  with Pre => X < Integer’Last;
```

- responsible: caller (must obey these conditions)
- verifier tries to prove every subprogram call

```
function Search (Arr : Arr_T; Key : Integer) return Natural
  with Pre => Sorted (Arr);

function Sorted (A : Arr_T) return Boolean is
  (A’Length < 2 or else
   (for all X in A’First .. A’Last - 1 =>
    (for all Y in X + 1 .. A’Last => A (X) <= A (Y))));
```

Don’t retype that...link to source is provided later!
Postcondition Contracts

- conditions that must hold true after subprogram finishes

  -- example:

  2 procedure Increment (X : in out Integer)
    with Pre  => X < Integer’Last,
    Post   => X = X’Old + 1;

- responsible: callee (must provide these conditions)
- verifier tries to prove that implementation actually produces these conditions
Postcondition Contracts

- conditions that must hold true after subprogram finishes

```
-- example:
procedure Increment (X : in out Integer)
  with Pre => X < Integer'Last,
  Post => X = X'0ld + 1;
```

- responsible: callee (must provide these conditions)
- verifier tries to prove that implementation actually produces these conditions

```
1 | -- our project:
  function Search (Arr : Arr_T; Key : Integer) return Natural
  with Pre  => Sorted (Arr),
     Post => (if Search’Result = 0 then True
         else Arr (Search’Result) = Key);
```
assign responsibilities

can be composed of
- any visible name in the scope of the subprogram
- any parameter of the subprogram

good types simplify contracts (e.g., use \texttt{Natural} instead of \texttt{Integer} when negative numbers are not allowed)
Even if proven free of errors, \( P \) may still produce errors when \( Q \) violates its postcondition!
Even if proven free of errors, \( P \) may still produce errors when \( Q \) violates its postcondition!
Even if proven free of errors, Q may still produce errors when P violates the precondition!
Even if proven free of errors, $Q$ may still produce errors when $P$ violates the precondition!
Hands-on Part III

⏰ 70min
Perfecting binary search

- make use of contracts
- time to fix your programs!
- (buggy) implementation so far:
  http://tinyurl.com/ycz2wk6l
Perfecting binary search (10min)

- make use of contracts
- time to fix your programs!
- (buggy) implementation so far: http://tinyurl.com/ycz2wk6l
- array index check might fail: Arr(mid) /= Key:
- loops are notoriously hard to analyze, solver needs hint
- add the following into loop to hint solver that low, high should always be in range:

  ```
  pragma Loop_Invariant (low in Arr'Range and high in Arr'Range);
  ```

- solver will verify this proposition and then use this fact to get rid of warning
Evaluation of Exercise

90% wrong?

Final solution: http://tinyurl.com/y8shbvo0

- extended postcondition: if result = 0, then key must not be in array
- 37 successfully discharged verification conditions in <1s
- no exceptions and definitely searching!
- all proved without testing
Advanced Topics

⏰ 82min
Combining Proof and Test

- sometimes proof may be too expensive
  - formal description of all properties may be very hard
  - Strong postconditions say “everything” about the subprogram; can’t always do that
Combining Proof and Test

- sometimes proof may be too expensive
  - formal description of all properties may be very hard
  - Strong postconditions say “everything” about the subprogram; can’t always do that
- divide application into pieces and apply either proof or testing to each piece
- contracts (and assertions) can be executed (“executable semantics”)
  - if program is compiled with assertions on
  - if they evaluate to False, then exception is raised
  - contracts can be exercised in a testing environment
Combining Proof and Test (Example)

Precondition ($\text{Tested}_\text{Proc}$) needs to be *proved*

Postcondition ($\text{Tested}_\text{Proc}$) needs to be *tested*
Support for Real-Time and Safety-Critical Software

Many things inherited from Ada

with Ada.Real_Time

- deterministic scheduling, priority-based interrupt handling, core assignment, preemption, resource sharing protocols
Support for Real-Time and Safety-Critical Software

Many things inherited from Ada

with Ada.Real_Time

- deterministic scheduling, priority-based interrupt handling, core assignment, preemption, resource sharing protocols
- pragma Profile (Ravenscar)
  - constrain tasking features to analyzable subset & patterns
  - easier schedulability analysis, memory boundedness, full determinism
- memory footprint starting at 2kB $\Rightarrow$ lower certification cost
Support for Real-Time and Safety-Critical Software

Many things inherited from Ada

with Ada.Real_Time

- deterministic scheduling, priority-based interrupt handling, core assignment, preemption, resource sharing protocols
- pragma Profile (Ravenscar)
  - constrain tasking features to analyzable subset & patterns
  - easier schedulability analysis, memory boundedness, full determinism
- memory footprint starting at 2kB ⇒ lower certification cost

Annex “Safety and Security”/”High-Integrity Systems”

- produce reviewable object code, initialize uninitialized variables to exceptions, option to block specific language features
Support for Real-Time and Safety-Critical Software

Many things inherited from Ada

with Ada.Real_Time

- deterministic scheduling, priority-based interrupt handling, core assignment, preemption, resource sharing protocols
- pragma Profile (Ravenscar)

- constrain tasking features to analyzable subset & patterns
- easier schedulability analysis, memory boundedness, full determinism
- memory footprint starting at 2kB ⇒ lower certification cost

Annex “Safety and Security”/”High-Integrity Systems”

- produce reviewable object code, initialize uninitialized variables to exceptions, option to block specific language features

Certifiable up to highest assurance levels, e.g., DO-178B Level A for commercial avionics (Level A: failure of software results in catastrophic consequences)
Using SPARK 2014 in Practice

The weather balloon story

- Ada & SPARK can be mixed!
  - a weather balloon that returns home
  - gradual adoption, partial verification
  - e.g., low-level drivers with pointers in Ada, high-level behavior in SPARK

- generate nightly reports
  - statistics about coverage, success and remaining work

- linking legacy code
  - use a C library from Ada or vice versa

- Code online: https://github.com/tum-ei-rcs/StratoX
Wrap-Up

⏰ 93min
Conclusion

Catching bugs was never that easy, but it comes with a price.

Next version of SPARK 2014 under development (pointers!).
Many Further Topics
Guides and Tutorials:

- *Ada95 Lovelace tutorial*, David A. Wheeler (who wrote that in his free time), http://www.adahome.com/Tutorials/Lovelace/lovelace.htm
References (2)

Background:

- *The Boeing 777 Flies on 99.9% Ada,*
- *Ada Glossary,* Bard S. Crawford,
  http://www.cs.uni.edu/~mccormic/AdaEssentials/glossary.htm
Technology behind the scenes:

- 
  RavenSPARK, R. Chapman, altran praxis,

- Static Analysis Tools Pass the Quals, AdaCore, Yannick Moy, 2014.


All online resources as of August 2017.
Extra Material (beyond 90’)

😊 95min
The SPARK 2014 Subset

Not within SPARK:
- pointers
- exception handlers
- full Ada tasking (e.g., complex synchronization)
- various minor things (e.g., no goto)

Not in Ada:
- certain SPARK-specific aspects (ignored by Ada compiler)
Concurrency = Ravenscar Profile

A “setting” that defines a deterministic subset of Ada’s tasking capabilities

- restrictions:
  - only certain types of synchronization mechanisms (e.g., protected objects)
  - scheduling policy: FIFO within priorities
  - static core assignment on multi-core systems
  - priority ceiling protocol
  - memory boundedness during link-time

- only subset is allowed in SPARK 2014 for tasking

Ravenscar Profile - Task Patterns

Patterns for Tasks

Each task must follow one of two implementation patterns:

1. **periodic**
   - infinite loop with single release point (`delay until`)

2. **sporadic**
   - infinite loop which takes up on an event (interrupt, blocking read from queue, ...)

Furthermore:

- number of tasks must be static
- tasks must not terminate
- no hierarchies of tasks
- . . .
Ravenscar Profile - Tasks, no hierarchy

- tasks are "all at same level"
- scope = package level
- master is always the system

```
with Bcontrol;
...procedure main is
  begin
    end main;

package body Bcontrol is
  ...task type taskX
  ...end taskX;
  ...
  X: taskX;
  ...
  end package;
```
Floating-Point Computations (‘‘digits’’)

General Floating point types

```haskell
    type Temperature is digits 18
        range  -173.15  ..  1.41679*10**32;

    type Mass is digits 7
        range  0.0  ..  1.0E35;
```

- very weak minimum requirements for float:
- `ideals` (non-numeric values such as `NaN`, `Inf`) can be excluded
- unspecified behavior: Overflow, underflow, zero-divide
- at least 6 digits of precision (RM §3.5.7); towards better portability
  - more digits: define own type with `digits` and optionally `range`
  - compiler rejects type if it cannot implement requested precision
- semantics of user types are not required to follow IEEE 754
Floating-Point Computations (2)

To get proof=reality, we must use IEEE-754 compliant FP computations

**Built-in Types:** Float and Long_Float

- floating-point hardware of target may or may not be used
  - GNAT compiler: will use FP hardware
  - non-GNAT: we can force the compiler to use FPU:

```plaintext
1 Temperature : Interfaces.IEEE_Float_32;
Mass : Interfaces.IEEE_Float_64;
```

- hardware may or may not be compliant
  - most FPU can/must be configured to be compliant (e.g., disable flush-to-zero)
  - GNAT: guarantees IEEE-754 semantics on x86 platforms, with flags `-msse2 -mfpmath=sse` to avoid extended 80bit precision

- complete model of Floating-Point arithmetic: RM §G.2.1.
Interval Arithmetic

Let $f$ be a numeric operation of $N$ arguments: $f(x_1, x_2, .., x_N)$. Then the corresponding interval operation $F(I_1, I_2, .., I_N)$ is defined as

$$F(I_1, I_2, .., I_N) = [l, u], \quad \text{where}$$

$$l = \inf(f(y_1, y_2, .., y_N))$$

$$u = \sup(f(y_1, y_2, .., y_N))$$

$$y_i \in I_i, \ i = 1..N$$

$\Rightarrow$ preserve accuracy instead of (usually) precision

**Example**

$$z = y_1 + y_2, \ \text{with} \ y_i \in [l_i, u_i]$$

$$\Rightarrow z \in [l_1 + l_2, u_1 + u_2]$$
Interval Arithmetics (2)

Ada library by Dmitry Kazakov, PhD (C.S., software architect)

- packages define types and overloads operations for tri-state logic, integers and floats.

```
function `'*'` (Left, Right : Interval) return Interval;
function "*" (Left : Interval; Right : Number) return Interval;
function "*" (Left : Number; Right : Interval) return Interval;
```

- internally using Float and Integer ⇒ implementation-defined results
- support rounding control (any rounding will produce intervals containing exact result)
- source code (GNU) available (e.g., to change to IEEE floats or fixed-point)

See http://www.dmitry-kazakov.de/ada/intervals.htm
Software with Sensitive Data

- one problem: when variable with sensitive data is deallocated, memory should be wiped, e.g.:

```
procedure Sensitive_Stuff is
  x : String := Get_Password();
beg
    -- ...
    x := (others => ' '); -- dead write, optimized out
end Sensitive_Stuff;
```

---

- How to wipe memory reliably in the owner?
- Martin Becker: Introduction to SPARK 2014 page 58 of 59
Software with Sensitive Data

- one problem: when variable with sensitive data is deallocated, memory should be wiped, e.g.:

```plaintext
procedure Sensitive_Stuff is
  x : String := Get_Password();
begin
  -- ...
  x := (others => ' '); -- dead write, optimized out
  -- at this point, memory allocated to x
  -- shall not hold password anymore
end Sensitive_Stuff;
```

- use `volatile ⇒ NO. Slower for all reads and writes; not allowed for local variables in SPARK 2014`

- `limited type (not copyable) ⇒ pass by ref enforced, solves ownership.`

- How to wipe memory reliably in the owner?
Software with Sensitive Data (2)

- `pragma Inspection_Point (from Annex H) ⇒` instructs compiler that object must be *inspectable* (i.e., not optimized out) at given program location

```haskell
x := (others => ''); -- dead write, optimized out
pragma Inspection_Point (x);
```

- dead store cannot be optimized out
- allows compiler optimization for all other accesses & provides traceability to registers!
- Further: GNATprove detects dead stores and issues warnings ⇒ annotate code with suppression = documentation of data sanitization
- paper (given below) gives a practical guideline, and discusses many more of this in SPARK 2014

---