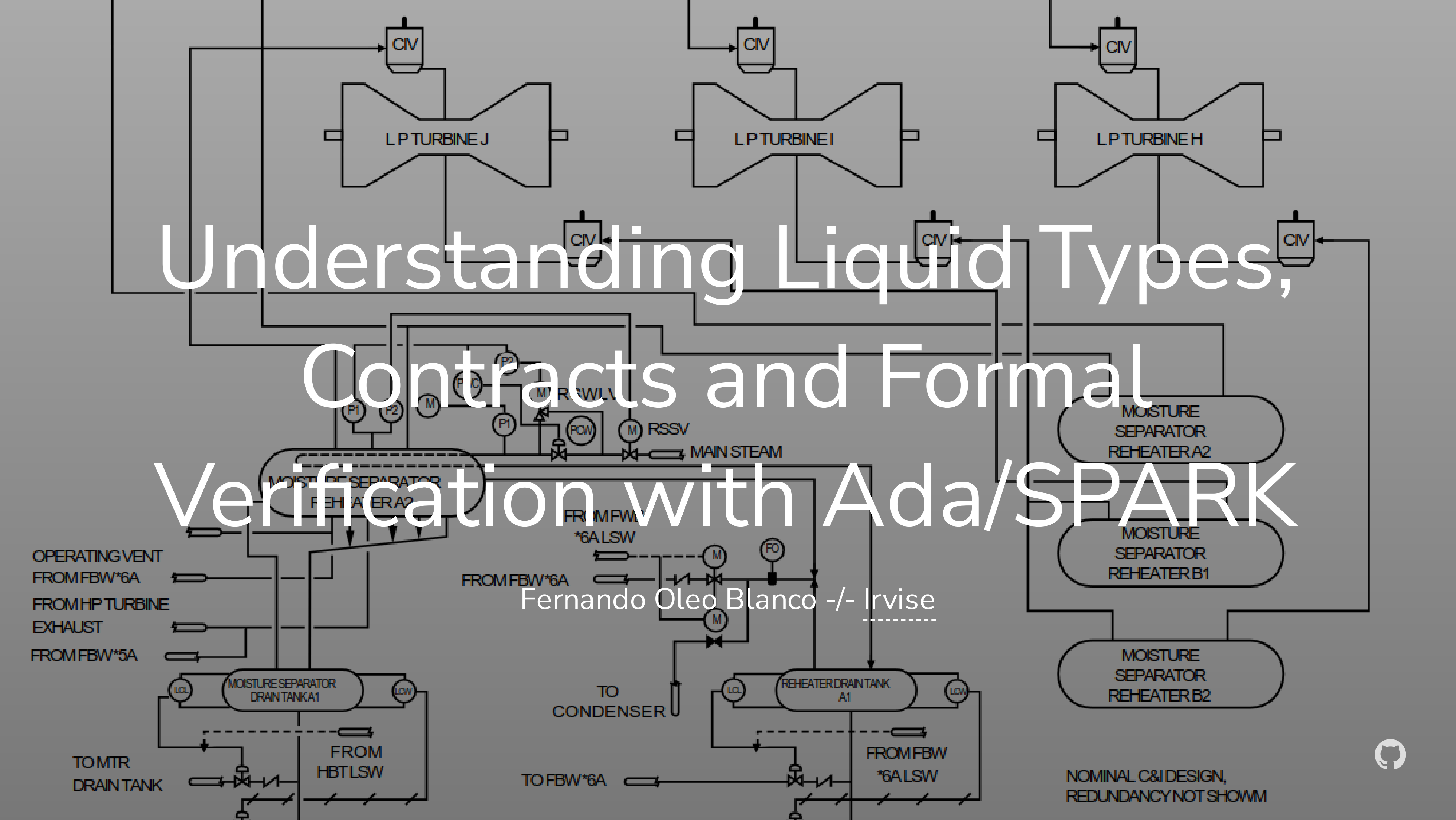


Understanding Liquid Types, Contracts and Formal Verification with Ada/SPARK

Fernando Oleo Blanco -/- Irvise

Topics and objectives

- Explain liquid types, contracts and proves...
  - ...in a practical and easy manner
  - Concepts will be universal (Haskell, Idris 2, The Rocq Prover, Frama-C, ML, F*, Lean4, Agda, ATS, Isabelle, TLA+)

Topics and objectives

- Explain liquid types, contracts and proves...
  - ...in a practical and easy manner
  - Concepts will be universal (Haskell, Idris 2, The Rocq Prover, Frama-C, ML, F*, Lean4, Agda, ATS, Isabelle, TLA+)
- Showcase Ada/SPARK
  - Little to no syntax will be explained
  - Only simple examples will be shown
  - Everything is valid... Ada 2012

Types!

Traditionally...

Computer/CPU/Memory-based information

```
int, float, char...
struct, union, void, nullptr
int *, int*, []
```

Or in newer Programming languages (`Zig` , `Rust` , `Jai`)

```
u8, f64, f16...
```

Types!

Traditionally...

Computer/CPU/Memory-based information

```
int, float, char...
struct, union, void, nullptr
int *, int*, []
```

Or in newer Programming languages (`Zig` , `Rust` , `Jai`)

```
u8, f64, f16...
```

Newer types focus on program flow and convey meaning, such as Rust’s

```
Optional<T> → Some<T> or None // See also Result<V, E>
```

Modern types are used to transmit real world meaning. But...

Types!

Traditionally...

Computer/CPU/Memory-based information

```
int, float, char...
struct, union, void, nullptr
int *, int*, []
```

Or in newer Programming languages (`Zig` , `Rust` , `Jai`)

```
u8, f64, f16...
```

Newer types focus on program flow and convey meaning, such as Rust's

```
Optional<T> → Some<T> or None // See also Result<V, E>
```

Modern types are used to transmit real world meaning. But...

Is this a passing test grade? `let Grade: u8 = 11;`

Ada’s Type System (fully abridged)

```
type Grade is range 0 .. 10; -- This is a Natural (Unsigned_Integer)
subtype Fail_Grade is Grade range 0 .. 4; -- Fully compatible with parent
```

Ada’s Type System (fully abridged)

```
type Better_Grade is delta 0.01 range 0.0 .. 10.0; -- Fixed point types FTW!
subtype Fail_Grade_V2 is Better_Grade range 0.0 .. 4.99;
```

Ada’s Type System (fully abridged)

```ada
type Better_Grade is delta 0.01 range 0.0 .. 10.0; -- Fixed point types FTW!
subtype Fail_Grade_V2 is Better_Grade range 0.0 .. 4.99;

-- Example
Failed : constant Boolean := 3.38 in Fail_Grade_V2; -- Returns True
```

Ada’s Type System (fully abridged)

```
type Better_Grade is delta 0.01 range 0.0 .. 10.0; -- Fixed point types FTW!
subtype Fail_Grade_V2 is Better_Grade range 0.0 .. 4.99;

-- Example
Failed : constant Boolean := 3.38 in Fail_Grade_V2; -- Returns True
```

Ada focuses on modelling real world data, the compiler does the heavy lifting

Play around with Ada in Compiler Explorer

Ada’s Type System (fully abridged)

```
type Better_Grade is delta 0.01 range 0.0 .. 10.0; -- Fixed point types FTW!
subtype Fail_Grade_V2 is Better_Grade range 0.0 .. 4.99;

-- Example
Failed : constant Boolean := 3.38 in Fail_Grade_V2; -- Returns True
```

Ada focuses on modelling real world data, the compiler does the heavy lifting

Play around with Ada in Compiler Explorer

What about more complex pieces of information?

- Even numbers? `2, 4, 6, 8 ...`
- Prime numbers? `2, 3, 5, 7, 11 ...`

Ada’s Type System (fully abridged)

```
type Better_Grade is delta 0.01 range 0.0 .. 10.0; -- Fixed point types FTW!
subtype Fail_Grade_V2 is Better_Grade range 0.0 .. 4.99;

-- Example
Failed : constant Boolean := 3.38 in Fail_Grade_V2; -- Returns True
```

Ada focuses on modelling real world data, the compiler does the heavy lifting

Play around with Ada in Compiler Explorer

What about more complex pieces of information?

- Even numbers? `2, 4, 6, 8 ...`
- Prime numbers? `2, 3, 5, 7, 11 ...`

Can we express the properties of data… as part of the type?

- Even numbers are (all the positive integer numbers) divisible by two
- Prime numbers are (all the positive integer numbers) which are only divisible by one and themselves

Liquid types!

Logically Qualified Types, aka, Types with Logic! Aka dependent types, etc...

Liquid types!

Logically Qualified Types, aka, Types with Logic! Aka dependent types, etc...

Lets express the logic/properties/etc as part of the type definition!

```
subtype Even is Natural...

subtype Odd is Natural...
```

Liquid types!

Logically Qualified Types, aka, Types with Logic! Aka dependent types, etc...

Lets express the logic/properties/etc as part of the type definition!

```
subtype Even is Natural with
  Dynamic_Predicate => Even mod 2 = 0;

subtype Odd is Natural with
  Dynamic_Predicate => Odd mod 2 = 1;
```

Liquid types!

Logically Qualified Types, aka, Types with Logic! Aka dependent types, etc...

Lets express the logic/properties/etc as part of the type definition!

```
My_Even_Var : Even := 2; -- Ok
Not_My_Even_Var : Even := 3; -- Not Ok! Compile with `-gnata` for runtime checks!
Is_Even : Boolean := 3 in Even; -- False
```

Liquid types!

Logically Qualified Types, aka, Types with Logic! Aka dependent types, etc...

Lets express the logic/properties/etc as part of the type definition!

```
My_Even_Var : Even := 2; -- Ok
Not_My_Even_Var : Even := 3; -- Not Ok! Compile with `-gnata` for runtime checks!
Is_Even : Boolean := 3 in Even; -- False
```

And what about primes?

```
type Prime is new Positive with
  Dynamic_Predicate => (for all Divisor in 2 .. Prime / 2 => Prime mod Divisor /= 0);
```

Liquid types!

Logically Qualified Types, aka, Types with Logic! Aka dependent types, etc...

Lets express the logic/properties/etc as part of the type definition!

```
My_Even_Var : Even := 2; -- Ok
Not_My_Even_Var : Even := 3; -- Not Ok! Compile with `-gnata` for runtime checks!
Is_Even : Boolean := 3 in Even; -- False
```

And what about primes?

```
type Prime is new Positive with
  Dynamic_Predicate => (for all Divisor in 2 .. Prime / 2 => Prime mod Divisor /= 0);
```

We can even use arbitrarily complex* functions in the predicate!

For more information see SPARK's User Manual on Type Contracts

Liquid types!

Logically Qualified Types, aka, Types with Logic! Aka dependent types, etc...

Lets express the logic/properties/etc as part of the type definition!

```
My_Even_Var : Even := 2; -- Ok
Not_My_Even_Var : Even := 3; -- Not Ok! Compile with `-gnata` for runtime checks!
Is_Even : Boolean := 3 in Even; -- False
```

And what about primes?

```
type Prime is new Positive with
  Dynamic_Predicate => (for all Divisor in 2 .. Prime / 2 => Prime mod Divisor /= 0);
```

We can even use arbitrarily complex* functions in the predicate!

For more information see SPARK's User Manual on Type Contracts

Boooooo!!!

Too academic and maths heavy! It is not useful in the real world!

A realistic case for liquid types

We have some measurement equipment (or maybe we are a high speed trading company)

```
type Day_Temperature is record
    High, Current, Low : Temperature;
end record;

Garden : Day_Temperature;

-- Somewher in the code...
Garden.Current := Get_Temperature("Garden");
--  ...and we move on
```

A realistic case for liquid types

We have some measurement equipment (or maybe we are a high speed trading company)

```
type Day_Temperature is record
    High, Current, Low : Temperature;
end record;

Garden : Day_Temperature;

-- Somewher in the code...
Garden.Current := Get_Temperature("Garden"); -- Bug! We forgot to potentially update High and Low!!
-- ...we get a silent data corruption error >:(
```

A realistic case for liquid types

We have some measurement equipment (or maybe we are a high speed trading company)

```
type Day_Temperature is record
    High, Current, Low : Temperature;
end record
  with Dynamic_Predicate => Day_Temperature.High >= Day_Temperature.Current and then
                            Day_Temperature.Current >= Day_Temperature.Low;

Garden : Day_Temperature;

-- Somewher in the code...
Garden.Current := Get_Temperature("Garden"); -- Runtime error!
-- ...we get a runtime exception (compile with -gnata) :/
```

A realistic case for liquid types

We have some measurement equipment (or maybe we are a high speed trading company)

```
type Day_Temperature is record
    High, Current, Low : Temperature;
end record
  with Dynamic_Predicate ⇒ Day_Temperature.High ⩾ Day_Temperature.Current and then
                           Day_Temperature.Current ⩾ Day_Temperature.Low;

Garden : Day_Temperature;

-- Somewher in the code...
Garden.Current := Get_Temperature("Garden"); -- Runtime error!
-- ...we get a runtime exception (compile with -gnata) :/
```

An array that is supposed to be sorted vs. one that actually is!

```
type Should_Be_Sorted is array (Index) of Integer; -- Are we sure it is sorted?

type Increasing_Ordered_Array is array (Index) of Integer
  with Dynamic_Predicate ⇒
    (for all I in Index ⇒ (if I < Index'Last then Ordered_Array(I) < Ordered_Array(I+1))); -- Nice!
```

Boooo, we can already do all of this since the 80s!

OOP example of a getter-setter pattern

```
class Garden_Day_Temperature {
private:
    Day_Temperature Garden_Temp;
public:
    float set(float Temperature) {
      ... // We make sure that this will be correct
    }
    float get() {
    }
};
```

Boooo, we can already do all of this since the 80s!

OOP example of a getter-setter pattern

```
class Garden_Day_Temperature {
private:
    Day_Temperature Garden_Temp;
public:
    float set(float Temperature) {
       ... // We make sure that this will be correct
    }
    float get() {
    }
};
```

Indeed but...

- OOP is a whole programming paradigm, hides data (`private`), requires opaque API (`set`, `get`, etc)
- **Just Types, Documentation (intent and meaning), Debugging/Instrumentation, Formal Verification**
  - OOP and liquid types can complement each other
- See `limited` keyword of Ada ;)

(Functional) Contracts

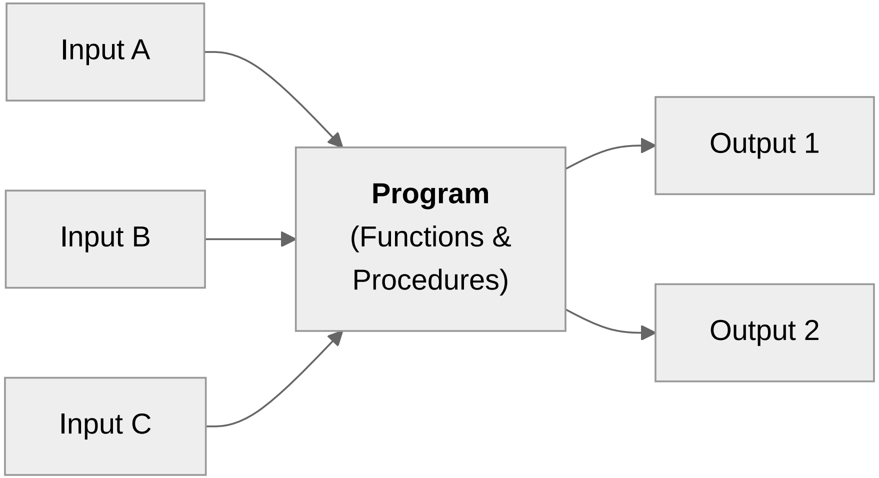
Liquid/dependent types focus on data. Contracts focus on the execution of code

Programs take inputs, process them and generate outputs

(Functional) Contracts

Liquid/dependent types focus on data. Contracts focus on the execution of code

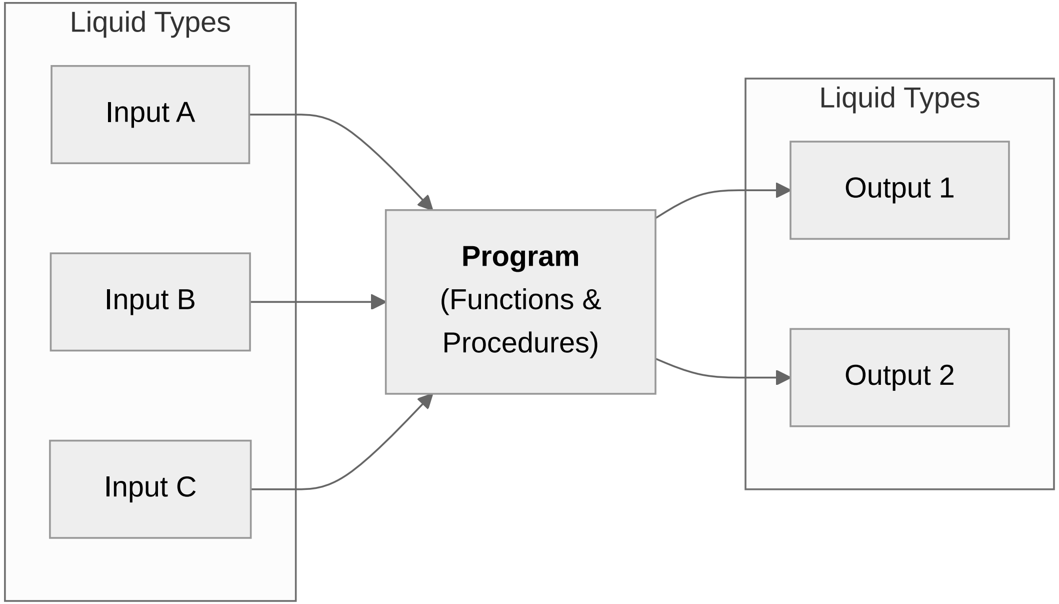
Programs take inputs, process them and generate outputs



(Functional) Contracts

Liquid/dependent types focus on data. Contracts focus on the execution of code

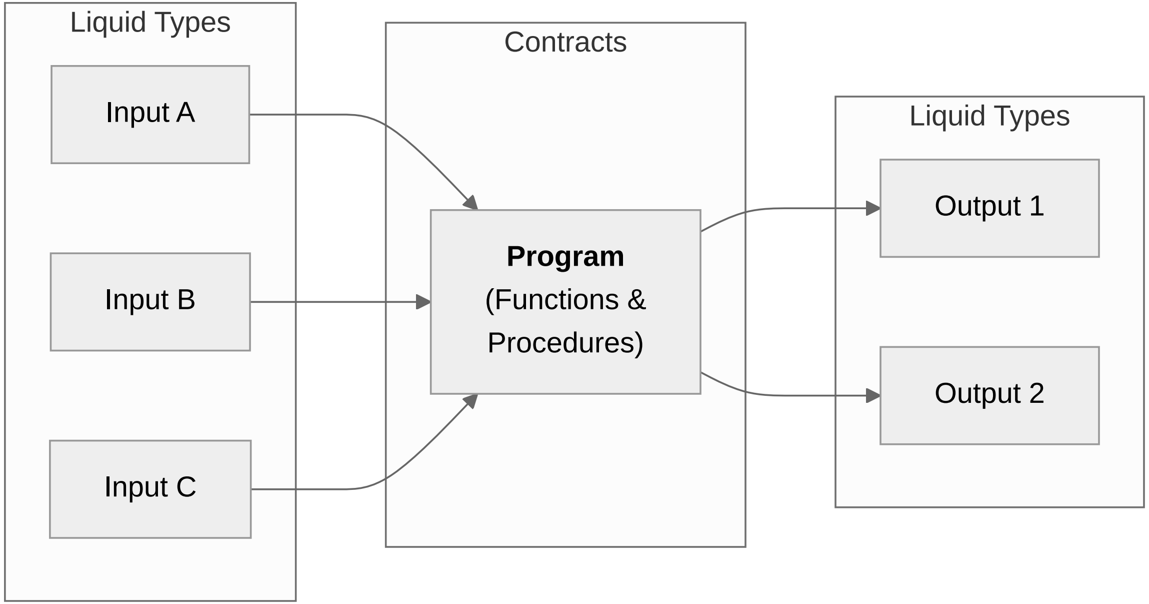
Programs take inputs, process them and generate outputs



(Functional) Contracts

Liquid/dependent types focus on data. Contracts focus on the execution of code

Programs take inputs, process them and generate outputs



Making sure functions do what we expect them to do

Adding meaning to the executable side of things

```
package Stack is

   procedure Push (V : Character);
   procedure Pop (V : out Character);
   procedure Clear;
   function Top return Character;

   function Full return Boolean is (Last = Max_Size);
   function Empty return Boolean is (Last < 1);
   function Size return Integer is (Last);

end Stack;
```

Making sure functions do what we expect them to do

Adding meaning to the executable side of things

```
package Stack is

   procedure Push (V : Character); -- What if the Stak is full!?
   procedure Pop (V : out Character); -- What if the Stak is empty!?
   procedure Clear; -- Did we implement this correctly?
   function Top return Character; -- Did we implement this correctly?

   function Full return Boolean is (Last = Max_Size);
   function Empty return Boolean is (Last < 1);
   function Size return Integer is (Last);

end Stack;
```

Making sure functions do what we expect them to do

Adding meaning to the executable side of things

```
package body Stack is

   procedure Push (V : Character) is
   begin
     if Full then
         raise Stack_Overflow with "Do not push data when Stack if full >:(!";
     end if;
     -- ...
   end Push;

   -- ...

end Stack;
```

Making sure functions do what we expect them to do

Adding meaning to the executable side of things

```
package Stack with SPARK_Mode => On is

   procedure Push (V : Character)
     with Pre  => not Full,             -- No need to check in the body
          Post => Size = Size'Old + 1;    -- Whatever the implementation is, this must hold
   procedure Pop (V : out Character)
     with Pre  => not Empty,            -- No need to check in the body
          Post => Size = Size'Old - 1;    -- Whatever the implementation is, this must hold
   procedure Clear
     with Post => Size = 0;              -- Whatever the implementation is, this must hold
   function Top return Character
     with Post => Top'Result = Tab(Last); -- Whatever the implementation is, this must hold

   function Full return Boolean is (Last = Max_Size);
   function Empty return Boolean is (Last < 1);
   function Size return Integer is (Last);

end Stack;
```

A couple of real world examples

```
--  From spark-containers-formal-vectors.ads
procedure Move (Target : in out Vector; Source : in out Vector)
with
  Global => null,
  Pre    => Length (Source) <= Capacity (Target),
  Post   => Model (Target) = Model (Source)'Old and Length (Source) = 0;
```

A couple of real world examples

```
--  From spark-containers-formal-vectors.ads
procedure Move (Target : in out Vector; Source : in out Vector)
with
                  --  We do not know how this is implemented...
  Global => null, --  but this function does not depend on any global state,
  Pre    => Length (Source) <= Capacity (Target),
                  --  there will be no buffer overflows
  Post   => Model (Target) = Model (Source)'Old and Length (Source) = 0;
                  --  and that the "Source" is emptied and no data can be lost
```

A couple of real world examples

```
procedure Insert --  From spark-containers-formal-unbounded_vectors.ads
  (Container : in out Vector;
   Before    : Extended_Index;
   New_Item  : Element_Type)
with Global ⇒ null, Pre ⇒ Length (Container) < Capacity (Container)
      and then Before in Index_Type'First .. Last_Index (Container) + 1,
     Post ⇒ Length (Container) = Length (Container)'Old + 1
      and M.Range_Equal       --  Elements located before Before in Container are preserved
            (Left  ⇒ Model (Container)'Old,
             Right ⇒ Model (Container),
             Fst   ⇒ Index_Type'First,
             Lst   ⇒ Before - 1)
      and Element_Logic_Equal --  Container now has New_Item at index Before
            (Element (Model (Container), Before), M.Copy_Element (New_Item))
      and M.Range_Shifted     --  Elements located after Before in Container are shifted by 1
            (Left   ⇒ Model (Container)'Old,
             Right  ⇒ Model (Container),
             Fst    ⇒ Before,
             Lst    ⇒ Last_Index (Container)'Old,
             Offset ⇒ 1);
```

Formal verification of software

Making sure that things work as expected... even before we compile/run them!

Can we be sure that our code is correct?

1. Correct the way we expect it to be (implicit behaviour)
2. Correct the way we wrote it to be (design)
3. Correct the way someone wants it to be (specification)

Formal verification of software

Making sure that things work as expected... even before we compile/run them!

Can we be sure that our code is correct?

1. Correct the way we expect it to be (implicit behaviour)
2. Correct the way we wrote it to be (design)
3. Correct the way someone wants it to be (specification)

Can we trust our code to not have issues/bugs?

- No memory errors (think of Rust's borrow checker)
- No program flow errors (forgetting to check a state)
- No unexpected arithmetic issues (over/underflows, division by zero)
- No type contracts (liquid types) errors
- No functional contracts errors
- No concurrency/parallel errors (data races)
- No incorrect handling of exceptions
- No runtime errors

Provers

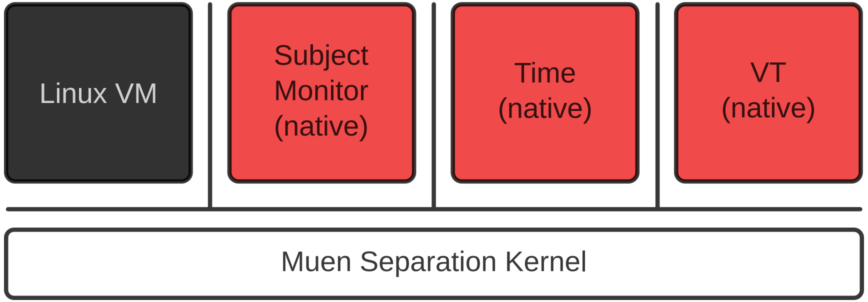
Software that takes a program as an input and analyse

- Static code analysis: the proof takes place **before the program has even been run!**
- Some provers are better at some things than others
  - Rust focuses mostly in memory correctness
  - TLA+ is widely used for concurrency and parallelism analysis
- Some provers are automatic (hands off), such as SPARK. Others are interactive, such as Rocq
- Some allow for customization of the prove
  - `unsafe` in Rust or `SPARK_Mode` and `--level=[1 .. 4]` in SPARK
- Not all provers produce an executable
  - Ada/SPARK is compiled…
  - …but the provers (Z3, Alt-Ergo, CVC5…) are just used for checking

But are these things actually used in real projects?!

Some examples of Ada/SPARK programs, but there are plenty more in other languages!

Muen Separation Microkernel

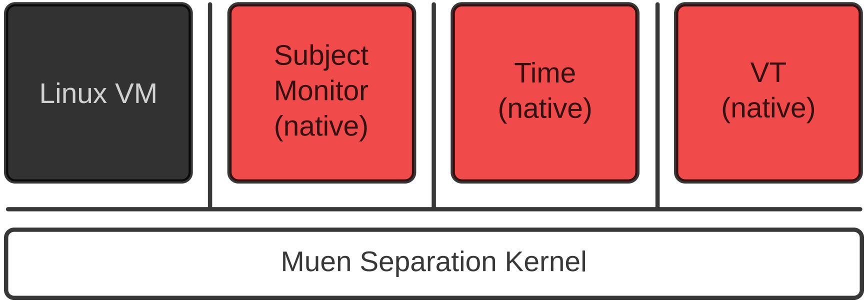


1. Formally verified in SPARK (and its components and drivers too!)
2. Used in telecomunication devices and cryptographic hardware (source)
3. Open source! (of course...)

But are these things actually used in real projects?!

Some examples of Ada/SPARK programs, but there are plenty more in other languages!

Muen Separation Microkernel



1. Formally verified in SPARK (and its components and drivers too!)
2. Used in telecomunication devices and cryptographic hardware (source)
3. Open source! (of course...)

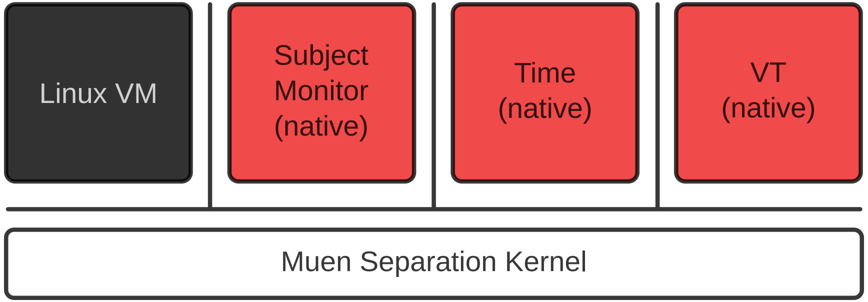
SPARKNaCl

1. Formally verified implementation of TweetNaCl
2. Initial version did catch a mistake of the original TweetNaCl release.
3. Designed to run on bare metal
4. Constant-time correct
5. Optimised for performance!

But are these things actually used in real projects?!

Some examples of Ada/SPARK programs, but there are plenty more in other languages!

Muen Separation Microkernel



1. Formally verified in SPARK (and its components and drivers too!)
2. Used in telecomunication devices and cryptographic hardware (source)
3. Open source! (of course...)

SPARKNaCl

1. Formally verified implementation of TweetNaCl
2. Initial version did catch a mistake of the original TweetNaCl release.
3. Designed to run on bare metal
4. Constant-time correct
5. Optimised for performance!

Others

1. jwx, SXML: formally verified JSON and XML parsers
2. RecordFlux, formally verified binary parsers, generators and protocols from state machines
3. QOI-SPARK, Quite-OK Image format in SPARK

Conclusion I

Liquid types and functional contracts

- Enhance types and execution with logic and properties
- Documentation of data and program flow: gives meaning and intent
- Instrumentation / Debuggability / Robustness of the code
- Compatible with other programming paradigms and designs
- Allow for formal verification

Formal verification (with SPARK)

- Full program analysis: no memory issues, logical problems, contracts/type violations, exception handling...
- Fully automated!
- A wonderful learning tool (and a strict one at that)
- Produces very high quality of software

Conclusion II

The language to rule (almost) them all... Ada...

Conclusion II

The language to rule (almost) them all... Ada...

Pros:

1. non Garbage-Collected (no GC)
2. low-level
3. high-level
4. runtime or proof-time
5. high-performance
6. large hardware support

   (whatever GCC and LLVM can handle in theory)
7. expressive (liquid types and contracts)
8. and ***very*** readable!

Conclusion II

The language to rule (almost) them all… Ada…

Pros:

1. non Garbage-Collected (no GC)
2. low-level
3. high-level
4. runtime or proof-time
5. high-performance
6. large hardware support
   (whatever GCC and LLVM can handle in theory)
7. expressive (liquid types and contracts)
8. and ***very*** readable!

The only language I know that ticks all these boxes

Conclusion II

The language to rule (almost) them all... Ada...

Pros:

1. non Garbage-Collected (no GC)
2. low-level
3. high-level
4. runtime or proof-time
5. high-performance
6. large hardware support
   (whatever GCC and LLVM can handle in theory)
7. expressive (liquid types and contracts)
8. and ***very*** readable!

The only language I know that ticks all these boxes

Cons:

1. not metaprogrammable (on purpose)
2. no macro system (on purpose)
3. lack of libraries (but you can help!)
4. small community (but you can help!)
5. lack of documentation (but you can help!)
6. advance features tend to be overlooked
   (but you can help!)

Thank you!

Questions?

Fernando Oleo Blanco -/- irvise@irvise.xyz

“There are only two kinds of languages: the ones people complain about and the ones nobody uses.”

— Bjarne Stroustrup

Extra: very low-level control with Ada

```
type BitArray is array (Natural range <>) of Boolean;
type Monitor_Info is record
        On     : Boolean;
        Count  : Natural range 0..127;
        Status : BitArray (0..7);
    end record;
```

Extra: very low-level control with Ada

```
with System; use System;
with System.Storage_Elements; use System.Storage_Elements;

type BitArray is array (Natural range <>) of Boolean with Pack; -- One bit per boolean
type Monitor_Info is record
        On     : Boolean;
        Count  : Natural range 0..127;
        Status : BitArray (0..7);
    end record
    with Size => 16, Volatile,
        Bit_Order => High_Order_First, Scalar_Storage_Order => High_Order_First;
        -- Big_Endian data representation regardless of the underlying hardware
```

Extra: very low-level control with Ada

```
-- ...
type BitArray is array (Natural range <>) of Boolean with Pack; -- One bit per boolean
type Monitor_Info is record
        On     : Boolean;
        Count  : Natural range 0..127;
        Status : BitArray (0..7);
     end record
     with Size => 16, Volatile, ... ;

for Monitor_Info use record
        On     at 0 range 0 .. 0;
        Count  at 0 range 1 .. 7;
        Status at 1 range 0 .. 7;
    end record; -- Define the bit position of the data, aka, representation clause!
```

Extra: very low-level control with Ada

```
--  ...
type BitArray is array (Natural range <>) of Boolean with Pack; -- One bit per boolean
type Monitor_Info is record
        On     : Boolean;
        Count  : Natural range 0..127;
        Status : BitArray (0..7);
     end record
     with Size => 16, Volatile, ... ;

for Monitor_Info use record
        On     at 0 range 0 .. 0;
        Count  at 0 range 1 .. 7;
        Status at 1 range 0 .. 7;
    end record; -- Define the bit position of the data, aka, representation clause!

My_MMIO_Thing : aliased Monitor_Info
  with Address => To_Address(16#6000_10A0#);
```