Extending Safe C Support In Leon

Marco Antognini

February 2017

Lab For Automated Reasoning And Analysis, EPFL

Talk Overview

GenC: safely converting Scala down to native C code.

1. Highlight of the key motivations for this work;
2. What is Leon and why you want it to verify programs;
3. Capabilities of *GenC* through case studies;
4. How MISRA compliance can be achieved.

Motivation

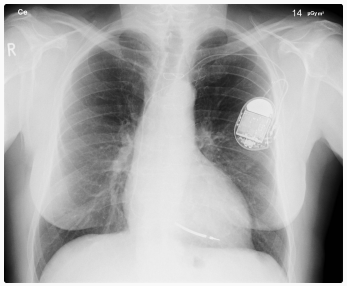
Software Nowadays

Hardware: more robust and efficient.

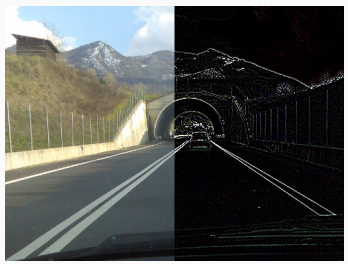
Solving a wider range of challenges.

Curiosity

BigDog



Pacemaker



Computer Vision

Image Source: Wikimedia

Fact:

Software complexity grows continuously.

Fact:

Software complexity grows continuously.

What about the quality of software?

CVE - Common Vulnerabilities and Exposures

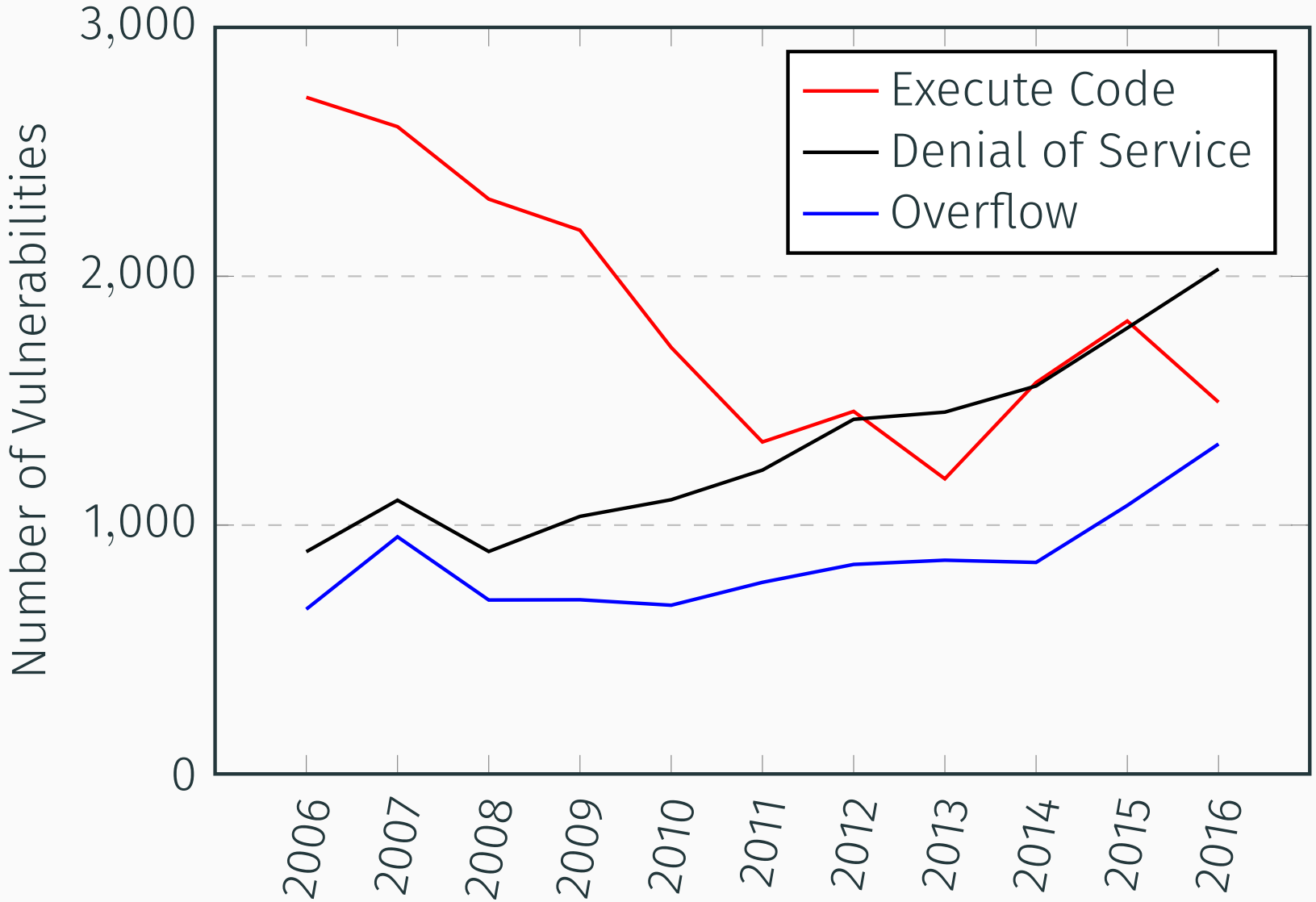


Figure: The 3 most frequent categories of bugs in the last 10 years.

Source: cve.mitre.org & cvedetails.com

CVE - Common Vulnerabilities and Exposures (cont.)

Credibility of the data?

Some numbers:

Life Span	Since 1999
Tickets	Over 80k (4k-8k by year in the last decade)
Actors	Adobe, Apache, Apple, Cisco, Google, Linux, Microsoft, Mozilla, ...

CVE - Common Vulnerabilities and Exposures (cont.)

Credibility of the data?

Some numbers:

Life Span	Since 1999
Tickets	Over 80k (4k-8k by year in the last decade)
Actors	Adobe, Apache, Apple, Cisco, Google, Linux, Microsoft, Mozilla, ...

***Overflows* are trending!**

Ranking by type	Overflows are third
Integer Overflows	$\sim$ 1.6% of total tickets
Buffer Overflows	$\sim$ 10% of total tickets

Critical Systems

Challenges

1. **No Failure Can Be Tolerated** (injury, death, money loss, ...)
2. **Embedded Devices** (spacecrafts, cars, medical implants, ...)
3. **Low Level Programming** (usually in C)

Critical Systems

Challenges

1. **No Failure Can Be Tolerated** (injury, death, money loss, ...)
2. **Embedded Devices** (spacecrafts, cars, medical implants, ...)
3. **Low Level Programming** (usually in C)

Addressing the challenges

1. **Verification & certification**;
2. Precise requirements for hardware;
3. Open question; C is known to be tricky.

C: Some Pitfalls

```c
int k = i << j; // when is this safe?
```

C: Some Pitfalls

```c
int k = i << j; // when is this safe?
```

- Range of **`int`**?
- Representation of **`int`**?
- Value of `j`?
- If `i` is signed, is `i << j` representable?
- Possible type cast and UB on assignment.
- ...

Certification: MISRA Guidelines for C

Motor Industry Software Reliability Association

Make C safer through a set of directives that:

- Put emphasis on *best practices*;
- Avoid triggering undefined, unspecified or implementation-defined behaviour;
- Restrict the language (e.g. no dynamic allocation, parts of *SL*, ...);
- Recommend specific arithmetic types & operations.

Do we have to write C code when targeting embedded devices?

Do we have to write C code when targeting embedded devices?

Yes and No.

Do we have to write C code when targeting embedded devices?

Yes and No.

Couldn't we write Scala code instead?

Do we have to write C code when targeting embedded devices?

Yes and No.

Couldn't we write Scala code instead?

Spoiler Alert: *Yes!*

Do we have to write C code when targeting embedded devices?

Yes and No.

Couldn't we write Scala code instead?

Spoiler Alert: *Yes!*

Can we detect overflows at compile time?

Do we have to write C code when targeting embedded devices?

Yes and No.

Couldn't we write Scala code instead?

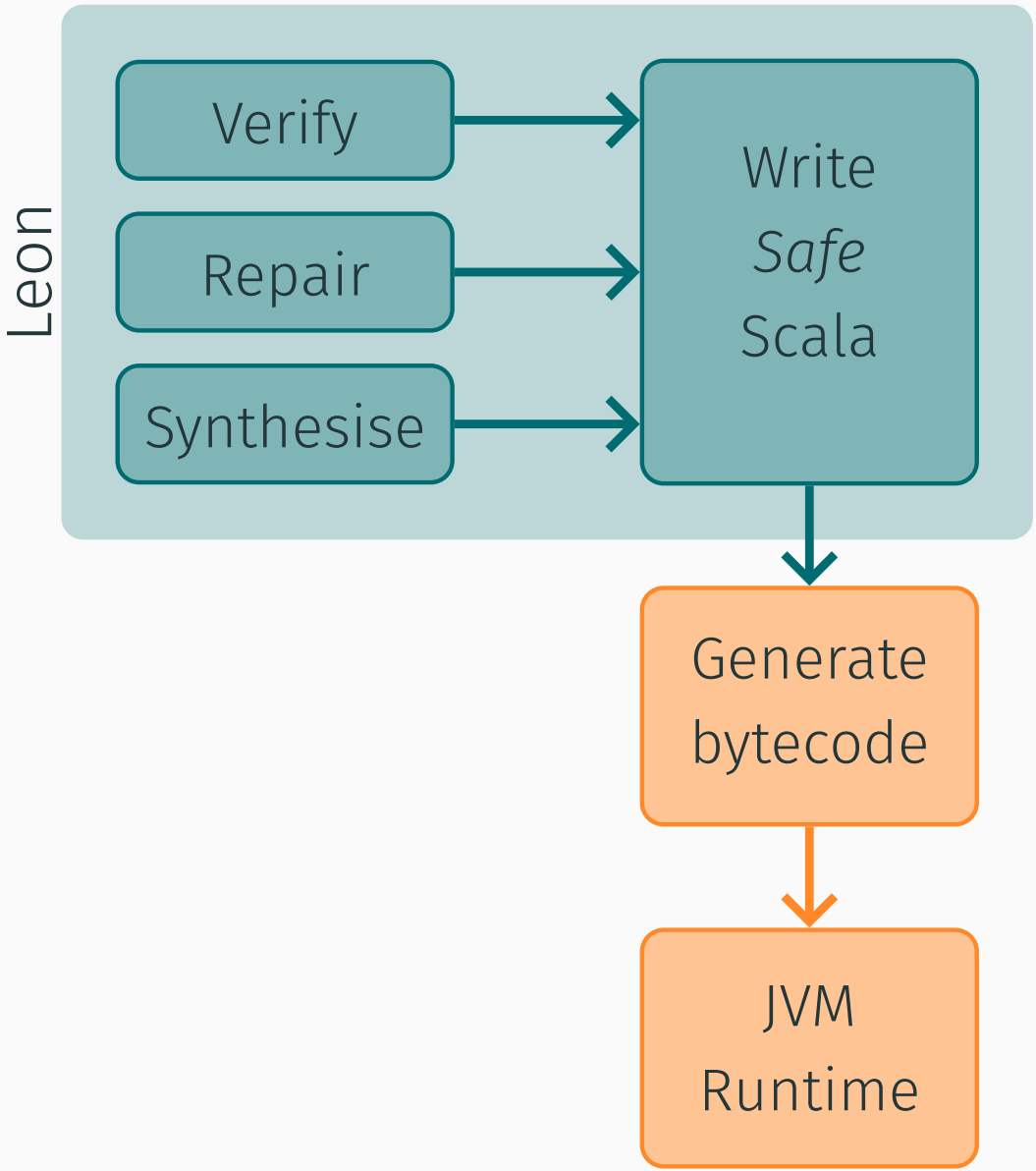
Spoiler Alert: *Yes!*

Can we detect overflows at compile time?

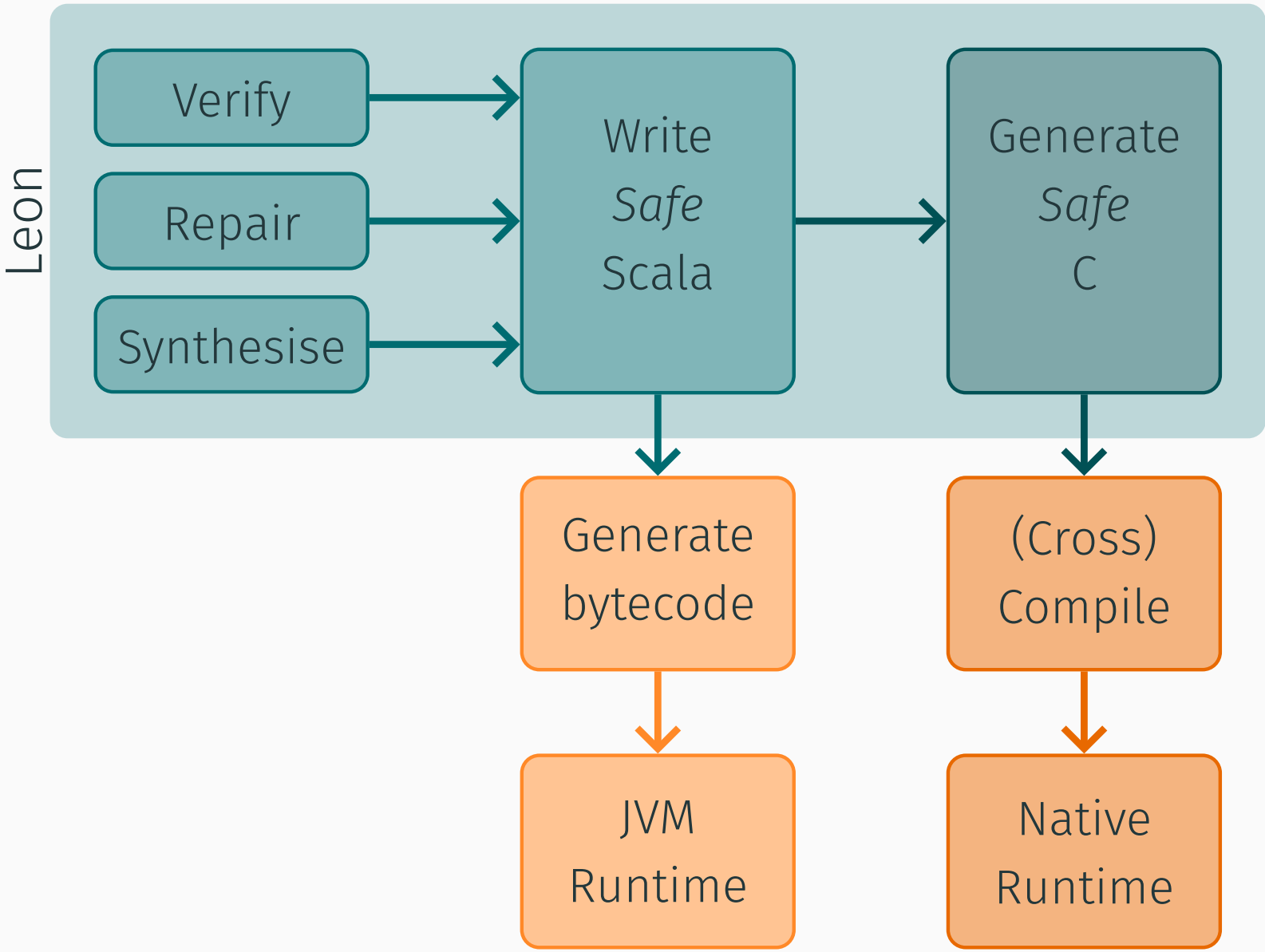
Yes.

Leon & *GenC*

Writing Safe Code With Leon



Writing Safe Code With Leon



Programming By Contract: Example A

```
def clamp(x: Int, down: Int, up: Int): Int = {
  require(down ⩽ up)
  max(down, min(x, up))
} ensuring { res ⟹ inRange(res, down, up) }
```

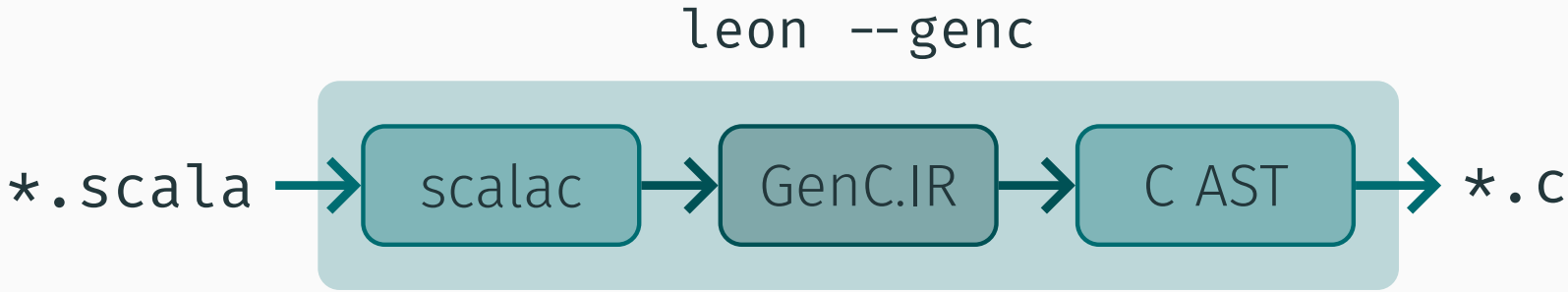
Programming By Contract: Example B

```
def skipBytes(fis: leon.io.FileInputStream, count: Int)
             (implicit state: leon.io.State): Boolean = {
  require(fis.isOpen && 0 ⩽ count)

  var i = 0
  var success = true

  (while (success && i < count) {
    val opt = fis.tryReadByte()
    success = opt.isDefined
    i += 1
  }) invariant (inRange(i, 0, count))

  success
}
```

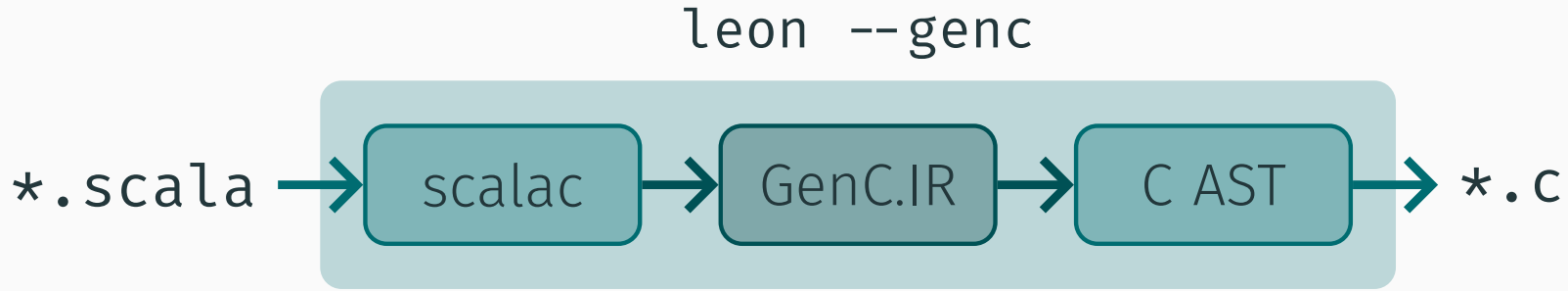
New Verification Features For Leon

- Support for `Byte` and integer cast operations.
- More `--overflow` checking:
  - `-Int.MinValue`
  - `Int.MinValue / -1`
- `--strict-arithmetic` checking:
  - Right hand side of `<< >> >>>` must be in $[0, 31]$;
  - `Int.MinValue % -1` undefined in C.
- Safe I/O API:
  - Provide interface for `[f]printf` and `[f]scanf`.

GenC



GenC



no UB
JVM-based
type safe
object
imperative

UB
native
imperative

functional  no first-class or higher-order functions

proofs  not translated

GenC: Memory Model

Mutability

JVM References $\equiv$ C Pointers.

No Aliasing $\Longrightarrow$ 1 owner of memory, deterministic lifetime.

Allocation

No dynamic memory allocation.

GenC: Supported Types

- Basic types & (ASCII) literals:

  `Unit Boolean Byte Int Char String`

- `Arrays[T]`: stack-allocated, can't be returned.
- `TupleN[T1, ..., TN]`.
- Classes:
  - `case class`, with mutable `var` fields;
  - Generics (no type erasure in Leon);
  - Inheritance with *tagged-union*;
  - "Enumeration" $\Longrightarrow$ `enum`;
  - No recursive types, e.g. `List`;
  - External types: `@cCode.typdef(alias, include)`.

GenC: Supported Functions

- Top-level or nested function;
- Free or member function;
- Generic or regular function;
- Overloaded or unique function;
- No higher-order function;
- External function: `@cCode.function(code, includes)`.

GenC: Supported Expressions

- Control flow: `if-else`, `while`, & pattern matching;
- Membership test & cast for object;
- Function call;
- Array access & update;
- Arithmetic operation:
  - Mimicking Java operators & integer promotion rules.
  - `+ - * / % & | ~ ^ && || << >>>`
- Normalised execution order.

Case Studies

LZW: Overview

Goal

1. Load/save file (binary I/O),
2. Compress/decompress data with a fixed amount of memory (based on dictionary manipulation).

LZW: Results

- Strings of **`Byte`** are represented using fixed-size **`Buffer`**s.
- Two implementations of **`Dictionary`**:
  A. **`Dictionary`** ≈ **`Array`**`[`**`Buffer`**`]` (nested arrays),
  B. **`Dictionary`** ≈ **`Array`**`[`**`Byte`**`]` (one array only).
- A. is *DRY* & faster but takes significantly more time to compile and uses more memory at runtime.
- Runtime bottleneck: dictionary lookup complexity.

Image Processing: Overview

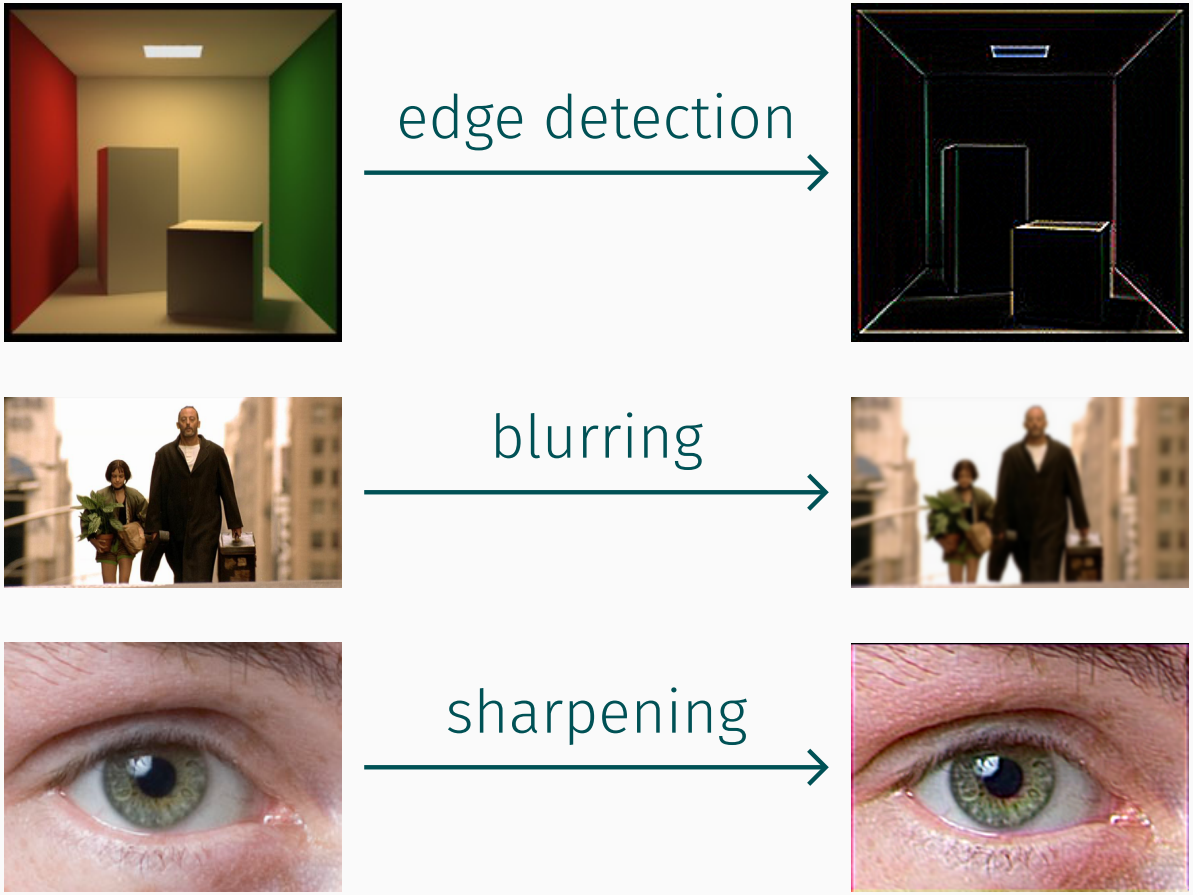
Goal

1. Load BMP file (I/O, parsing),
2. Apply some kernel (matrix convolution) onto the image,
3. Save the resulting image as BMP (more I/O, conversion).

Image Processing: Using Kernels

```
// Sharpen
val kernel = Kernel(size = 5, scale = 25, Array(
  -1, -1, -1, -1, -1,
  -1,  2,  2,  2, -1,
  -1,  2,  8,  2, -1,
  -1,  2,  2,  2, -1,
  -1, -1, -1, -1, -1
))

def processImage(src: Image): Status = {
  val dest = createImage(src.w, src.h)
  kernel.apply(src, dest)
  saveImage(fos, dest)
}
```

Image Processing: Kernels



All outputs are produced through Leon/*GenC*.

MISRA: Working Toward Compliance

Directive 1.1

Any implementation-defined behaviour on which the output of the program depends shall be documented and understood.

Defining a Standard C Environment

Directive 1.1

Any implementation-defined behaviour on which the output of the program depends shall be documented and understood.

Requirements for the C99 implementation

- The availability of **`int8_t`**, **`int32_t`** and **`uint32_t`** types;
- Converting **`int32_t`** $\longleftrightarrow$ **`uint32_t`** works as expected;
- A *byte* has 8 bits;
- No other language extensions or requirements,
- No use of assembly instructions.

Limiting Runtime Errors

Directive 4.1

Run-time failures shall be minimized.

Limiting Runtime Errors

Directive 4.1

Run-time failures shall be minimized.

Avoiding runtime errors

1. *Arithmetic errors* are detected with `--strict-arithmetic`;
2. *Pointer arithmetic* is not used;
3. *Function contract violations* and
4. *Array overflows* are detected when verifying the code;
5. *Pointers dereferencing* are safe under our memory model;
6. No *dynamic memory allocations* are used.

Rule Subsumed Under Verification: Array Overflows

Rule 18.1

A pointer resulting from arithmetic on a pointer operand shall address an element of the same array as that pointer operand.

Rule Subsumed Under Verification: Array Overflows

Rule 18.1

A pointer resulting from arithmetic on a pointer operand shall address an element of the same array as that pointer operand.

```
def swap(data: Array[Int],
         i: Int, j: Int) {
  val tmp = data(i)
  data(i) = data(j)
  data(j) = tmp
}
```

Rule Subsumed Under Verification: Array Overflows

Rule 18.1

A pointer resulting from arithmetic on a pointer operand shall address an element of the same array as that pointer operand.

```
def swap(data: Array[Int],
         i: Int, j: Int) {
  val tmp = data(i)
  data(i) = data(j)
  data(j) = tmp
}
```

```
static void swap_0(array_int32 data_0,
                   int32_t i_3, int32_t j_0) {
  int32_t tmp_0 = data_0.data[i_3];
  data_0.data[i_3] = data_0.data[j_0];
  data_0.data[j_0] = tmp_0;
}
```

Rule Subsumed Under Verification: Array Overflows

Rule 18.1

A pointer resulting from arithmetic on a pointer operand shall address an element of the same array as that pointer operand.

```
def swap(data: Array[Int],
         i: Int, j: Int) {
  val tmp = data(i)
  data(i) = data(j)
  data(j) = tmp
}
```

```
static void swap_0(array_int32 data_0,
                   int32_t i_3, int32_t j_0) {
  int32_t tmp_0 = data_0.data[i_3];
  data_0.data[i_3] = data_0.data[j_0];
  data_0.data[j_0] = tmp_0;
}
```

Example of Verification Report

Function	Kind	Result	Time
swap	array usage	⚠ invalid	0.025

The following inputs violate the VC:

```
data := Array(0)
i    := -2147483648
```

Other Considerations

Verification identifies potential problems before execution.

GenC carefully crafts the C code to respect many rules.

Other Considerations

Verification identifies potential problems before execution.

GenC carefully crafts the C code to respect many rules.

But *GenC* does not claim to replace MISRA:

- Function results should be used;
- There should be no unreachable or dead code;
- Recursive functions should be avoided;
- ...

20 rules are left to the user of the 159 overall rules.

Conclusion

Summary

- More verification tools for Leon.
- A significant fragment of Scala safely translatable to C.
- A simple memory model is possible when no aliasing.
- Most of MISRA is guaranteed to be satisfied.
- Case studies showed we can write software with *GenC*!

Future Works – Beside *GenC* Itself

MISRA

Linter: warn user about (more) rule violations.

Performance: HashMap Case Study

Scala vs C comparison.

Resource Inference

Combine *GenC* and Orb to infer memory & time metrics.

More Numerical Types & Libraries

`Short`, **`Long`**, **`FixedPoint`**, **`ArrayList`**, ...

Thank You!

LLVM IR vs C

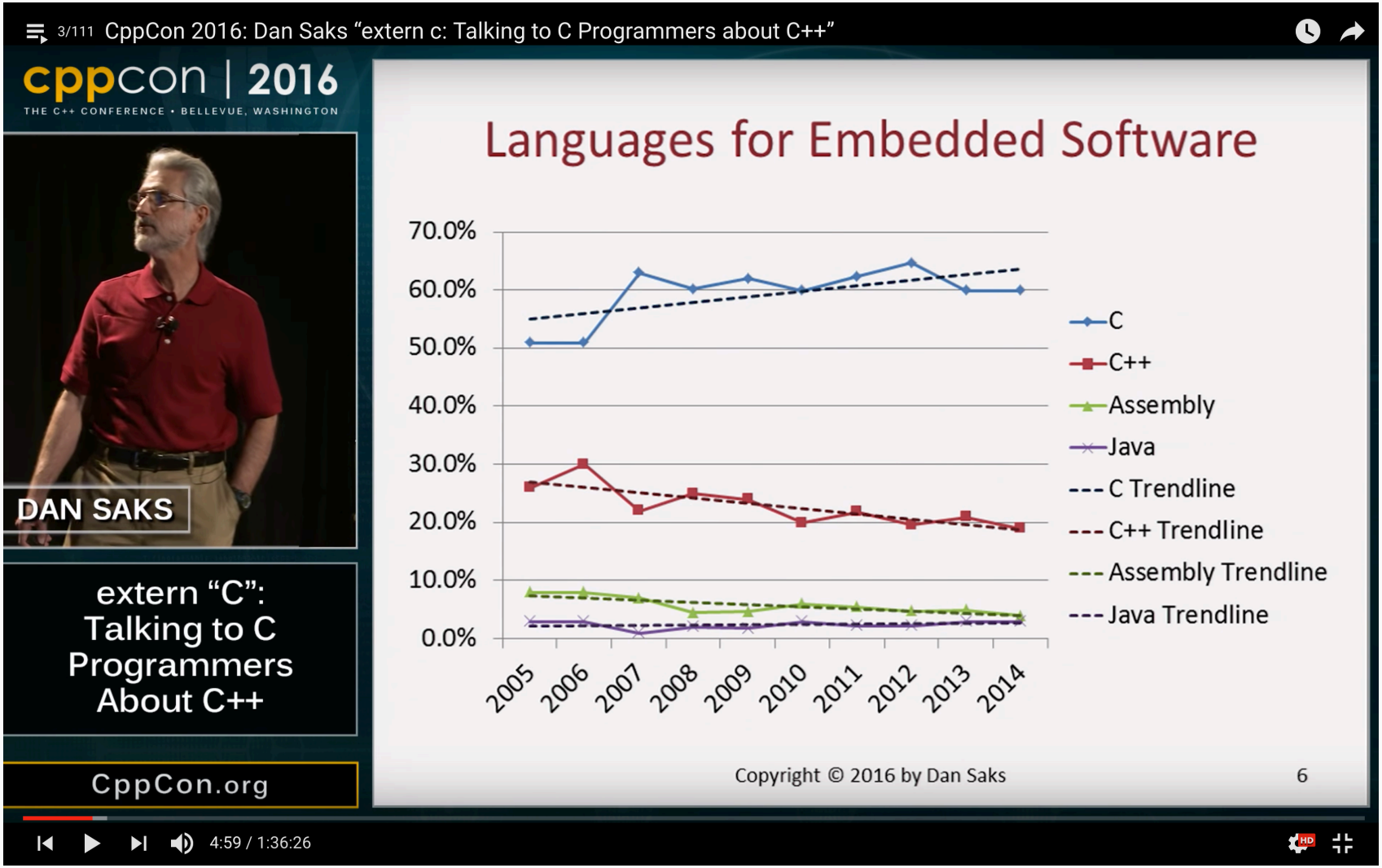
Why C99 was chosen?

- C is widely used;
- Many compilers for C for many hardware architectures;
- Prior experience in C;
- MISRA “compatible”.

Is LLVM IR a good alternative?

- Probably yes: getting popular;
- MISRA principles could be “translated”;
- Lower level means more opportunity for optimisation;
- Probably more strictly defined than C.

Language Shares



https://youtu.be/D7Sd8A6_fYU

Programming By Contract: Example A, Fixed

```
def clamp(x: Int, down: Int, up: Int): Int = {
  require(down ⩽ up)
  max(down, min(x, up))
} ensuring { res ⇒
  inRange(res, down, up) &&
  (inRange(x, down, up) ⟹ (res == x))
}
```

Supported Features

Basic Types

- Basic Types:

Unit	Boolean	Byte	Int	Char	String
void	bool	int8_t	int32_t	char	char*

- And their respective literals[1].

[1] ASCII literals only

Arrays

- `Array[T]` is mapped into

```
typedef struct { T* data; int32_t length; }
        array_T;
```

Arrays

- `Array[T]` is mapped into

```
typedef struct { T* data; int32_t length; }
        array_T;
```

- An array needs two *allocation points*:

```
val a = Array(0, 1, 2, 3)
```

is translated into

```
int32_t leon_buffer_0[4] = { 0, 1, 2, 3 };     // (1)
array_int32 a = (array_int32) {                // (2)
  .data = leon_buffer_0, .length = 4
};
```

Arrays

- `Array[T]` is mapped into

```
typedef struct { T* data; int32_t length; }
        array_T;
```

- An array needs two *allocation points*:

```
val a = Array(0, 1, 2, 3)
```

  is translated into

```
int32_t leon_buffer_0[4] = { 0, 1, 2, 3 };     // (1)
array_int32 a = (array_int32) {                // (2)
  .data = leon_buffer_0, .length = 4
};
```

- Stack-allocated arrays cannot be returned from functions!
- *Variable Length Array* (VLA) are supported with a warning.

Case Classes

- Generic case class without inheritance and `TupleN` are mapped to `struct`:

```
case class Pair[A, B](x: A, y: B)
```

when `A` = `Int` and `B` = `Boolean` is translated into

```
typedef struct { int32_t x_0; bool y_0; }
        Pair_0_int32_bool;
```

Case Classes

- Generic case class without inheritance and `TupleN` are mapped to `struct`:

  ```
  case class Pair[A, B](x: A, y: B)
  ```

  when `A` $=$ `Int` and `B` $=$ `Boolean` is translated into

  ```
  typedef struct { int32_t x_0; bool y_0; }
          Pair_0_int32_bool;
  ```

- Generics $\approx$ C++ templates:
  - No type erasure in Leon;
  - Instantiated at compile time for every combination of type parameters needed.

Case Classes

- Generic case class without inheritance and **`TupleN`** are mapped to **`struct`**:

  ```
  case class Pair[A, B](x: A, y: B)
  ```

  when `A` $=$ **`Int`** and `B` $=$ **`Boolean`** is translated into

  ```
  typedef struct { int32_t x_0; bool y_0; }
          Pair_0_int32_bool;
  ```

- Generics $\approx$ C++ templates:
  - No type erasure in Leon;
  - Instantiated at compile time for every combination of type parameters needed.

- Mutable fields (**`var`**) are available too.

Inheritance (1/3)

- When inheritance is involved, a *tagged-union* is used:

```
abstract class P
case class C1(/* ... */) extends P
/* ... */
case class CN(/* ... */) extends P
```

is translated into the following types:

```
typedef struct { /* ... */ } C1_0;
/* ... */
typedef struct { /* ... */ } CN_0;
typedef enum { tag_C1_0, /* ... */, tag_CN_0 }
        enum_P_0;
typedef union { C1_0 C1_0_v; /* ... */ CN_0 CN_0_v; }
        union_P_0;
typedef struct { enum_P_0 tag; union_P_0 value; }
        P_0;
```

Inheritance (2/3)

- Special case for enumerations: only an **enum** is used.

```
abstract class Status
case class Success()     extends Status
case class OpenError()   extends Status
case class ReadError()   extends Status
case class DomainError() extends Status
/*  ...  */
```

is mapped into

```
typedef enum {
  tag_Success_0,
  tag_OpenError_0,
  tag_ReadError_0,
  tag_DomainError_0,
  /*  ...  */
} enum_Status_0;
```

Inheritance (3/3)

- Leon supports generics only at the top level; so does *GenC*.
- No heap allocation implies no recursive types, e.g. **`List`**.

Inheritance (3/3)

- Leon supports generics only at the top level; so does *GenC*.
- No heap allocation implies no recursive types, e.g. **`List`**.
- Types are *lifted*:

```
val c = C1(0)
val s = OpenError()
```

is transpiled into

```
P_0 c_5 = (P_0) {
  .tag = tag_C1_0,
  .value = (union_P_0) { .C1_0_v = (C1_0) { .x_1 = 0 } }
};

enum_Status_0 s_19 = tag_OpenError_0;
```

Functions

- Generic, overloaded or nested functions & methods are all lifted to top level functions:

```
def foo[A, B](optA: Option[A], optB: Option[B],
              altA: A, altB: B): (A, B) = {
  def nested[C](opt: Option[C], alt: C): C =
    opt getOrElse alt

  (nested(optA, altA), nested(optB, altB))
}
```

  is translated into 3 functions[2] for `A` $\neq$ `B`,
  resulting in 17 x86-64 instructions (GCC 6.3 `-O1`).

[2] `Option[T].getOrElse` is inlined

Functions

- Generic, overloaded or nested functions & methods are all lifted to top level functions:

```
def foo[A, B](optA: Option[A], optB: Option[B],
              altA: A, altB: B): (A, B) = {
  def nested[C](opt: Option[C], alt: C): C =
    opt getOrElse alt

  (nested(optA, altA), nested(optB, altB))
}
```

  is translated into 3 functions[2] for `A` $\neq$ `B`, resulting in 17 x86-64 instructions (GCC 6.3 `-O1`).

- No higher-order functions.

[2] `Option[T].getOrElse` is inlined

Expressions

- Control Flow:
  - **`if`**, **`else if`**, **`else`**,
  - **`while`**,
  - Pattern matchings are first converted into **`if`**-**`else`**.
- Membership Test & Cast:
  - `isInstanceOf[T]` using tag of *tagged-union*,
  - `asInstanceOf[T]` by accessing the **`union`**.
- Function call.
- Array access & update.
- Arithmetic operations:
  - Mimicking Java operators & integer promotion rules.
  - `+ - * / % & | ~ ^ && || << >>>`
- All that with normalisation in mind.

Library Support

- Plug in external C libraries using:
  - `@cCode.function(code, includes)`
  - `@cCode.typedef(alias, include)`
  - `@cCode.drop()`
- Example:
  - I/O API: provide interface for `[f]printf` and `[f]scanf`.

Image Processing: Code Example

```
case class Kernel(size: Int, scale: Int, kernel: Array[Int]) {

  def apply(src: Image, dest: Image): Unit = {
    require(src.w == dest.w && src.h == dest.h)

    val size = src.w * src.h
    var i = 0

    (while (i < size) {
      dest.r(i) = apply(src.r, src.w, src.h, i)
      dest.g(i) = apply(src.g, src.w, src.h, i)
      dest.b(i) = apply(src.b, src.w, src.h, i)

      i += 1
    }) invariant (inRange(i, 0, size))
  }

}
```

Image Processing: Code Example (bis)

```scala
case class Kernel(size: Int, scale: Int, kernel: Array[Int]) {
  private def apply(channel: Array[Byte], width: Int, height: Int, index: Int): Byte = {
    require( /* ... */ )

    // Get the color component at the given position
    def at(col: Int, row: Int): Int = { /* ... */ } ensuring { inRange(_, 0, 255) }

    val mid = size / 2
    val i = index % width ; val j = index / width

    var res = 0 ; var p   = -mid
    (while (p ⩽ mid) {
      var q = -mid
      (while (q ⩽ mid) {
        val kcol = p + mid ; val krow = q + mid
        val kidx = krow * size + kcol

        // Here, the += and * operation could overflow
        res += at(i + p, j + q) * kernel(kidx)

        q += 1
      }) invariant (inRange(q, -mid, mid + 1))
      p += 1
    }) invariant (inRange(p, -mid, mid + 1))

    clamp(res / scale, 0, 255).toByte
  }
}
```