FROM VERIFIED FUNCTIONS TO SAFE C CODE

Marco Antognini

Fall 2015

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

MOTIVATION

MOTIVATION

- Software keeps getting more complex.
- Hardware gets cheaper and more robust.
- For some industry, development cost is high mostly because of software verification process.
- Failure can have disastrous impact on human lives or huge monetary loss.

$\longrightarrow$ We need tools to verify complex software.

LEON?

- Leon offers tools to reduce this cost.
- But! It's JVM-based hence useless for very small devices (e.g. pacemakers, spacecraft, ...)
- Need native code.
- That's why C-like languages are widely used in the industry.

VERIFIER FOR C?

- Why not but low level, verbose and error prone.
- Modern C++ or D might solve many issues from C…
- …but no good verifiers exist.

VAL GENC = (CAST) LEON

GENC

- Use Leon to verify, repair and synthesis code using high level features of Scala*.

GENC

- Use Leon to verify, repair and synthesis code using high level features of Scala*.
- Translate code into equivalent C99 code.

GENC

- Use Leon to verify, repair and synthesis code using high level features of Scala*.
- Translate code into equivalent C99 code.
- Compile it with your favourite C compiler for your destination architecture.

GENC

- Use Leon to verify, repair and synthesis code using high level features of Scala*.
- Translate code into equivalent C99 code.
- Compile it with your favourite C compiler for your destination architecture.
- And run it natively on your hardware.

Supported Features

TYPES

`Int` $\longrightarrow$ `int32_t` `Boolean` $\longrightarrow$ `bool` `Unit` $\longrightarrow$ **`void`**

TYPES

`Int` ⟶ `int32_t`   `Boolean` ⟶ `bool`   `Unit` ⟶ `void`

`TupleN[T1, T2, ..., TN]` is generated using some kind of templates. E.g. `(Boolean, Int)` is:

```
typedef struct __leon_tuple_bool_int32_t_t {
  bool    _1;  // padding
  int32_t _2;
} __leon_tuple_bool_int32_t_t;
```

TYPES

`Int` ⟶ `int32_t`   `Boolean` ⟶ `bool`   `Unit` ⟶ **`void`**

`TupleN[T1, T2, ..., TN]` is generated using some kind of templates. E.g. `(Boolean, Int)` is:

```
typedef struct __leon_tuple_bool_int32_t_t {
  bool    _1;  // padding
  int32_t _2;
} __leon_tuple_bool_int32_t_t;
```

Similarly, `Array[T]` is templatised. E.g. `Array[(Boolean, Int)]` is:

```
typedef struct __leon_array___leon_tuple_bool_int32_t_t_t {
  __leon_tuple_bool_int32_t_t* data;   // not owning the memory
  int32_t                      length;
} __leon_array___leon_tuple_bool_int32_t_t_t;
```

Those arrays live on the stack $\longrightarrow$ cannot return them.

VLA (variable-length array) are used when needed.

```scala
def foo(size: Int, value: Int) { val a = Array.fill(size)(value) }
```

Those arrays live on the stack ⟶ cannot return them.

VLA (variable-length array) are used when needed.

```
def foo(size: Int, value: Int) { val a = Array.fill(size)(value) }
```

is translated into

```
void foo0(int32_t const size0, int32_t const value0) {
  int32_t __leon_vla_buffer0[size0]; // actual memory alloc
  for (int32_t __leon_i1 = 0; __leon_i1 < size0; ++__leon_i1) {
    __leon_vla_buffer0[__leon_i1] = value0;
  }
  __leon_array_int32_t_t const a3 =
    { .length = size0, .data = __leon_vla_buffer0 };
}
```

FUNCTIONS

Nested functions are extracted with their context.

```
def foo(x: Int) = {
  def bar(y: Int) = x * y
  bar(2)
}
```

Nested functions are extracted with their context.

```
def foo(x: Int) = {
  def bar(y: Int) = x * y
  bar(2)
}
```

is translated into

```
int32_t bar0(int32_t const* x0, int32_t const y6)
{ return (*x0) * y6; }

int32_t foo0(int32_t const x0)
{ return bar0((&x0), 2); }
```

FUNCTIONS — BIS

```scala
def foo(x: Int) = {
  def bar(y: Int) = {
    def fun(z: Int) = x * y + z
    fun(3)
  }
  bar(2)
}
```

FUNCTIONS — BIS

```
def foo(x: Int) = {
  def bar(y: Int) = {
    def fun(z: Int) = x * y + z
    fun(3)
  }
  bar(2)
}
```

is translated into

```
int32_t fun0(int32_t const* x0, int32_t const* y6, int32_t const z9)
{ return (*x0) * (*y6) + z9; }

int32_t bar0(int32_t const* x0, int32_t const y6)
{ return fun0(x0, (&y6), 3); }

int32_t foo0(int32_t const x0)
{ return bar0((&x0), 2); }
```

IF CONSTRUCT

Unlike in Scala, **if** statements don't return a value. Hence

```scala
def foo(x: Int) {
  val b =
    if (x >= 0) true else false
}
```

is translated into

```c
void foo0(int32_t const x0) {
  bool b0; // no const here
  if (x0 >= 0) { b0 = true; }
  else { b0 = false; }
}
```

IF CONSTRUCT

Unlike in Scala, **if** statements don't return a value. Hence

```
def foo(x: Int) {
  val b =
    if (x >= 0) true else false
}
```

is translated into

```
void foo0(int32_t const x0) {
  bool b0; // no const here
  if (x0 >= 0) { b0 = true; }
  else { b0 = false; }
}
```

```
def foo(x: Int) =
  if (x >= 0) true else false
```

is translated into

```
bool foo1(int32_t const x0) {
  if (x0 >= 0) { return true; }
  else { return false; }
}
```

WHILE CONSTRUCT

```scala
def dummyAbs(a: Array[Int]) {
  var i = 0;
  while (i < a.length) {
    a(i) = if (a(i) < 0) -a(i) else a(i)
    i = i + 1
  }
}
```

WHILE CONSTRUCT

```
def dummyAbs(a: Array[Int]) {
  var i = 0;
  while (i < a.length) {
    a(i) = if (a(i) < 0) -a(i) else a(i)
    i = i + 1
  }
}
```

is translated into

```
void dummyAbs0(__leon_array_int32_t_t const a0) {
  int32_t i9 = 0;
  while (i9 < a0.length) {
    if (a0.data[i9] < 0) { a0.data[i9] = -a0.data[i9]; }
    else { a0.data[i9] = a0.data[i9]; }
    i9 = i9 + 1;
  }
}
```

SUBEXPRESSIONS EXECUTION ORDER

- C standard is much more lax when it comes to execution order.
- Hence the translation has to do some kind of normalisation to ensure the same behaviour.
- This applies to function calls, operators, and blocks.

SUBEXPRESSIONS EXECUTION ORDER — BIS

```
def test4(b: Boolean) = {
  var i = 10;
  var c = 0;
  val f = b && !b // false
  val t = b || !b // true

  while ({ c = c + 1; t } &&
         i > 0 ||
         { c = c * 2; f }) {
    i = i - 1
  }

  i == 0 && c == 22
}
```

SUBEXPRESSIONS EXECUTION ORDER — BIS

```scala
def test4(b: Boolean) = {
  var i = 10;
  var c = 0;
  val f = b && !b // false
  val t = b || !b // true

  while ({ c = c + 1; t } &&
         i > 0 ||
         { c = c * 2; f }) {
    i = i - 1
  }

  i == 0 && c == 22
}
```

```cpp
bool test40(bool const b0) {
  int32_t i10 = 10;
  int32_t c2 = 0;
  bool const f26 = b0 && (!b0);
  bool const t8  = b0 || (!b0);

  while (true) {
    c2 = c2 + 1;
    if (t8 && i10 > 0) { i10 = i10 - 1; }
    else {
      c2 = c2 * 2;
      if (f26) { i10 = i10 - 1; }
      else { break; }
    }
  }

  return i10 == 0 && c2 == 22;
}
```

Recap'

- ▷ Basic types
- ▷ Tuples & stack allocated arrays
- ▷ Nested functions
- ▷ If & While constructs
- ▷ Subexpressions orders

WHAT'S NEXT

FUTURE'S WORK

- Support for non-recursive data type (`case class`).
- A case study about image processing.

FUTURE'S WORK

- Support for non-recursive data type (`case class`).
- A case study about image processing.

- Support for floating point types (`float`, `double`).
- Support for `BigInt`, `Real`, e.g. using GMP.
- Support for heap allocated arrays, e.g. using `malloc`.

QUESTIONS?