From Verified Functions to Safe C Code

Marco Antognini

January 2016

Optional Master Semester Project under the supervision of
Viktor Kuncak
Lab for Automated Reasoning and Analysis LARA - EPFL

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

1 Overview

As software becomes more and more complex we need ways to ensure it works according to its specification. While hardware is more and more reliable and cheaper, even for extreme environments such as space exploration (e.g. spacecraft) or medical device (e.g. pacemaker), software development costs substantially increase when programs have to be robust and safe. Many companies spend a significant part of their budget in making sure their software won't put people's life in danger or result in a huge monetary loss.

Leon[1] works at solving this issue for programs written in (a subset of) Scala by providing tools to verify contracts, repair erroneous implementations or even synthesis code. However, Scala being based on JVM runtime, such tools are close to worthless for many companies that run their software on very small devices: the lack of memory and CPU resources prevents running virtualised code on such hardware. This explains why those systems are written in low-level, C-like languages.

We therefore extend Leon to generate standard C99 code from a subset of Scala in order to benefit from the high-level features of this language and reduce the development cost of low-level software while avoiding using error-prone languages.

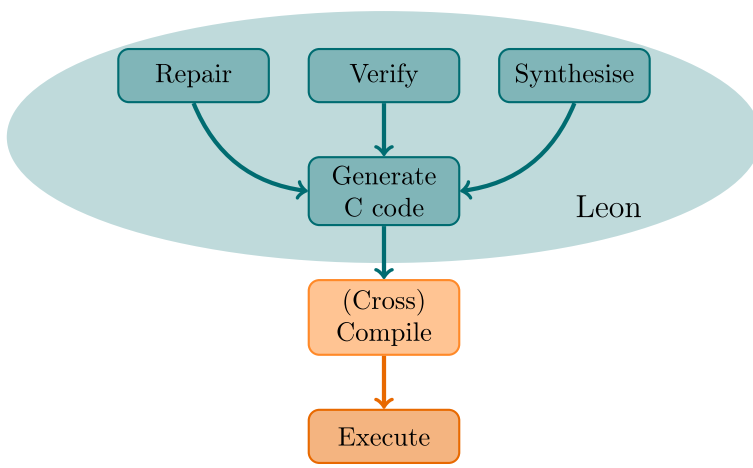

Figure 1: Development Process

The overall, high level development pipeline is illustrated in Figure 1. The first step is to generate a valid and verified Scala program using Leon. The input source code should contains only one top-level Scala **`object`** that represents the program to be converted. Then the program can be converted into an equivalent C99 code through the *GenC* phase using the `--genc` command line option. Please refer to Leon's manual for the complete invocation details as well as

[1] `http://leon.epfl.ch`

detailed explanation on how to verify, repair or synthesis programs, and more.

The produced code can then be compiled using any standard-compliant C99 compiler – for example Clang[2] or GCC[3] – to generate a native and optimised assembly code for specific hardware architectures. Then the compiled program can be shipped to the desired hardware and executed as usual.

2 Implementation Details

In this section we give an overview of the generated C99 code and its structure, plus some general insight on the implementation of *GenC* inside Leon.

2.1 Generated C99 Code

The translated code follows a strict structure imposed by the difference between the Scala and C languages. The main reason for this structure is that, in Scala, the order of type and function definitions have little importance from a compiler perspective but in C every type of function has to be at defined, or at least declared, before being used. That is why the outputted code starts by `#include`'ing the necessary headers, then forward-declares every custom data types – such as tuples, case classes and arrays types presented in Section 3.1 – so that their declarations, which follows next, can refer to each other if needed. After data types, the same strategy is applied to user-defined functions: first they are declared, in any order, and then their bodies are fully defined.

A complete example of generated safe C code is presented in Appendix B.

2.2 GenC Phase in Leon

The *GenC* pipeline is made of several independent phases, that were already defined in Leon, as shown in Figure 2 where the `PreprocessingPhase` is detailed in the right column with disabled sub-phases represented by grey boxes. The roles of the three first phases are to extract the Scala Abstract Syntax Tree (AST) from the input source code, pre-process it to add Leon-specific information and transform it into a usable format for the `GenerateCPhase`. This latter phase will produced a C AST from its input so that the last phase can pretty print it into a given file.

The source code related to *GenC* is stored in the `src/main/scala/leon/genc/` direction of Leon's git repository. In particular, the `CAST.scala` file contains a minimalistic AST for the C language for the needs of this project, while `CConverter.scala` holds most of the source code responsible for the conversion from the Scala AST to the C one.

[2] `http://clang.llvm.org/`
[3] `https://gcc.gnu.org/`

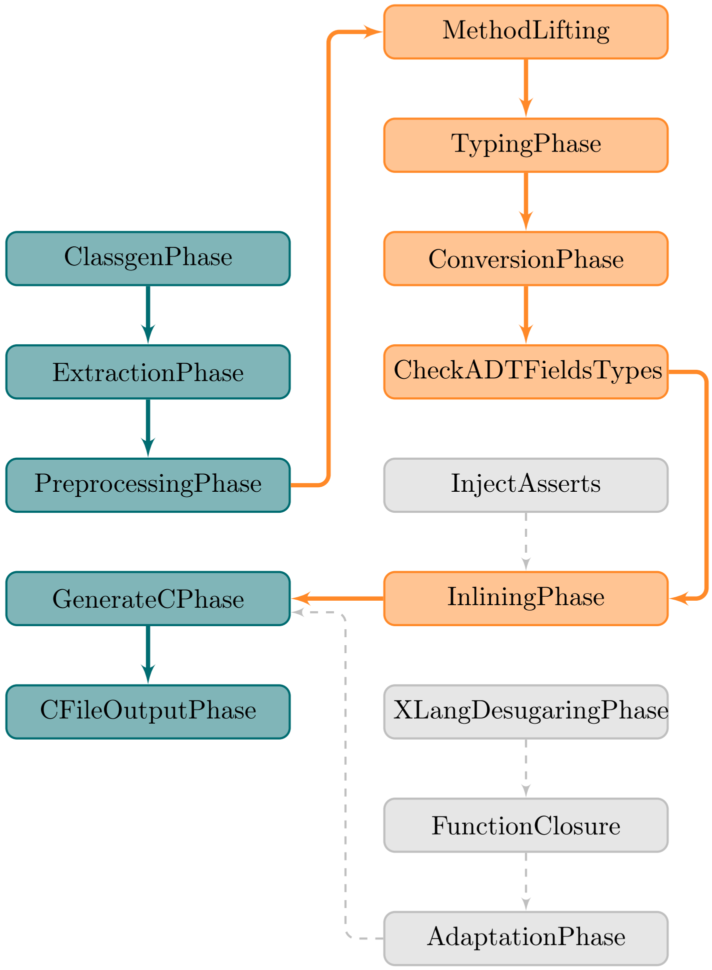

Figure 2: *GenC* Pipeline when invoking Leon with `--genc`; grey elements are disabled

3 Supported Features

The current state of the *GenC* phase supports programs using basic boolean and integer types, tuples, arrays, non-recursive case classes, functions and nested functions, `if` and `while` constructs and guarantees the expressions execution order in the generated C99 program to be consistent with the execution model of Scala.

3.1 Types

In this section we give a summary of the different Scala types supported by *GenC* with their corresponding C99 types. We also detail the limitations on the current state of Scala to safe C translation on the mentioned types.

3.1.1 Basic Types

Table 1 reports the three basis types that are currently supported. In Scala, `Int` is subject to the same overflow behaviour as the 32-bit integer type in C. Currently, Leon doesn't verify that no overflow occurs but rather uses this behaviour to prove – or disprove – that some other properties hold, such as the index used to read a value from an array is not out-of-bound.

Scala	C99
`Unit`	`void`
`Boolean`	`bool` (from `<stdbool.h>`)
`Int` (32-bit integer)	`int32_t` (from `<stdint.h>`)

Table 1: Supported basic types

The respective literals, such as `true`, `false`, and numbers are supported as well, of course. Note that when extracting the Scala AST, the compiler might convert some literals format, such as hexadecimal to numbers in base 10. However, the Scala literal `()` for the `Unit` type has no equivalent in C99 and therefore is simply ignored.

3.1.2 Tuples

In addition to those basic types and their corresponding literals, *GenC* also supports generic tuples of size N. In Scala, tuples are represented with different types according to their size: for example `(true, 1)` is of type `Tuple2[Boolean, Int]` while `(2, false, 3)` is of type `Tuple3[Int, Boolean, Int]`. In order to represent each possible `TupleN[T1, ..., TN]` type in C99, without having access to Scala's generics, we generate a new C structure for every combination of types `T1, ..., TN` used in the original program, as a C++ templatised structure would have generated at compilation time. The name of such structures is defined by the concatenation of the combined types' name with `__leon_tuple` and `_t` as prefix and postfix, respectively. The field of those structures are simply matching the one from the source language. Accessing elements of a tuple, say `_1`, is equivalently trivial in C.

Listing 1 shows the equivalent C99 code that represents `Tuple2[Boolean, Int]` type. In this particular case, having a boolean field generates some padding in the memory layout of the structure.

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

Listing 1: Example of tuple type in C99

Note that tuples can be made of other tuples, arrays or case classes at will. However, the restrictions on arrays as discussed in Section 3.1.5 also apply on tuples containing arrays.

3.1.3 Case Classes

GenC also has a basic support for case classes. Currently it is restricted to non-recursive types, without inheritance involved, where all member are values. And, as with tuples, such data types can hold arrays as field with the restriction mentioned in Section 3.1.5. Essentially, case classes are mapped to a C structure which fields are one-to-one C equivalent of the Scala original type. Listing 2 shows the C99 equivalent of declaring `case class Pixel(r: Int, g: Int, b: Int)`.

```c
typedef struct Pixel0 {
  int32_t const r0;
  int32_t const g0;
  int32_t const b0;
} Pixel0;
```

Listing 2: Example of case class in C99

Instantiating a case class is usually done using the compiler-defined companion object of the same name. In C99, since there is no notion of constructor, we have to rely on *designated initialiser lists*. Listing 3 illustrates how `val red = Pixel(255, 0, 0)` is converted into C99.

```c
Pixel0 const red0 = (Pixel0) { .r0 = 255, .g0 = 0, .b0 = 0 };
```

Listing 3: Example of case class initialisation in C99

3.1.4 Arrays

Similarly to Scala's tuples, we handle the generic `Array[T]` type by generating specific C99 structures for each `T` involved in the source program. However, as the size of an array in Scala is not encoded into its type, we cannot allocate a specific amount of memory with the matching C99 type definition. Instead, we create those structures with two fields: one representing the length of the array and the other being a pointer to the memory allocated for the array.

We can delay the memory allocation to the instantiation site of an array. The actual memory allocation is discussed in the next section. Listing 4 shows the corresponding type definition for a `Array[(Boolean, Int)]` as an example.

```
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;
```

Listing 4: Example of tuple type in C99

Independently on how an array is actually allocated, accessing one of its elements is done through the usual C-array index access on its `data` field. Table 2 shows a simple example of this.

Scala	`def foo(a: Array[Int]) = a(42)`

C99	`int32_t foo0(__leon_array_int32_t_t const a0) {`
	`  return a0.data[42];`
	`}`

Table 2: Example of array access

3.1.5 Memory Model

Since Scala's arrays have a fixed size once created, we don't have to implement complex mechanism to properly resize arrays. Instead, we have to deal with two main operations: allocating memory and deallocating it when appropriate. The same is true for both tuples and case classes.

Because Leon currently doesn't allow aliasing on objects, the lifetime of variables is trivial to deduce as only three scenarios can happen. First of all, an objects can be a function parameter in which case the current function can only observe its state without mutating it and no deallocation should happen there as the current function doesn't own the memory. Secondly, it can be created in a given function `f` and then returned. In this case, we can consider the caller of `f` to own the memory and therefore it falls in fact in the last case. Finally, a function can own an object if it doesn't return it or have acquired it through one of its parameters. Hence, in this case, the function is responsible to deallocate it when no longer needed. This point in time is in the worst case when the function returns. It is therefore safe to deallocate any owned piece of memory at this precise moment.

The order in which the memory is deallocate needs not to be constrained by the allocation order as no notion of destructor, as defined in C++ for example, is present in Scala and therefore no "late" access to objects can happen.

Due to the restrictions imposed on some specific domains such as aircraft software or some tiny embedded system regarding memory safety, we decided that, in the first version of *GenC*, no manual allocation should be done and instead only stack-allocated objects can be created. This is of course a very restrictive decision as it implies returning arrays is not possible without moving the allocation site to the caller when the array is returned from a function. In fact, if the size of a returned array is not known at compile time it becomes even more arduous to handle. A solution to this specific problem could be to inline the called function directly in the generated C code.

However, in order to have a working implementation decently quickly we decided to forbid returning arrays and types containing arrays. This restriction doesn't apply to other tuples or case classes thanks to the no-aliasing policy of Leon which implies that copying a returned variable of fixed size is valid. Note that, due to the fact that the length of arrays is not encoded in the type, we cannot relax this restriction to arrays of fixed length as this information is not available to the caller.

With that in mind, we can deal with array allocation in two ways. First, for fix-sized array we can use regular C-array as shown in Table 3: a buffer of the corresponding size is allocated on the stack and then an instance of the corresponding array type is created with the `data` field pointing to the allocated buffer and the `length` field being initialised appropriately. This buffer will get automatically deallocated when the program reaches the end of its scope. When the size of the array is only known at runtime we can rely on Variable Length Array (VLA) to create the memory buffer representing the array as shown in Table 4: instead of being statically defined, the size of the buffer is determined dynamically but the its lifetime remains the same.

3.2 Variables

Among the many differences between Scala and C is the immutability idiom used in functional languages compared to the constness expressed in imperative languages. For a basic type `T`, such as **`Int`** or **`Boolean`**, having in Scala **`val`** `x: T = ...` corresponds to having `T` **`const`** `x = ...` in C.

Scala	`val a = Array(1, 2, 3);`

C99	`int32_t __leon_buffer0[3] = { 1, 2, 3 };`
	`__leon_array_int32_t_t const a3 = {`
	`  .length = 3,`
	`  .data   = __leon_buffer0`
	`};`

Table 3: Example of fix-sized array

Scala	<pre>def foo(size: Int, value: Int) {
	  val a = Array.fill(size)(value)
	}</pre>

C99	<pre>void foo0(int32_t const size0, int32_t const value0) {
	  int32_t __leon_vla_buffer0[size0];
	  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
	  };
	}</pre>

Table 4: Example of runtime allocated array

Regarding arrays, even if they are declared as **`val`**, the elements of the arrays can still be mutated[4]. This is reflected in C as well by using a pointer to represent the beginning of the memory allocated for the array: whether or not an instance of `__leon_array_T_t` is marked with **`const`**, the `data` field gives read and write access to the elements of the array.

As for compound data types such as tuples or case classes, we can simply mark as **`const`** any instance of those types as *GenC* currently support only immutable case classes and because tuples are defined to be immutable in Scala.

3.3 Functions

The support for functions in *GenC* is relatively complete from an imperative point of view. That is, first-order functions are supported as well as nested functions but higher-order functions are not. Additionally, since strings of characters are not yet implemented, the main function should be defined in Scala as `def main: Int = ...`.

During the extraction of the Scala AST, the variable, function and class identifiers are renamed as to avoid any ambiguity. This is why two functions originally named `foo` in the input source code, were they declared in different scopes or simply overloads, would be renamed into `foo0` and `foo1` in the AST. The direct consequence of this is that function overloading is directly supported by *GenC* without extra work.

[4]Leon currently restricts, due to aliasing issues, the update of arrays to the ones declared locally. Hence, passing an array as parameter make it read only.

3.3.1 Nested Functions

In order to support nested functions, which don't exist in vanilla C, we have to outline nested functions without modifying the scope of variables. The idea is to add extra parameters to extracted functions to extend the function's context to include what was available in the original source code.

However, in C parameters are pass-by-value and therefore copied while in Scala arguments are either pass-by-name, which *GenC* does not support at the moment, or pass-by-reference (to use in C-terminology). This in itself means that if a nested function mutates a variable created in the outer function, the effect using pass-by-value in C to share the variable would not allow the extracted function to have the desired side-effect. We therefore have to use pointer to simulate the pass-by-reference nature of Scala parameters when calling a nested function and dereference pointers when accessing those variables.

Table 5 illustrates how we can handle 1-level nested function. For deeper nested functions we have an additional issue: parameters that were already pointers-to-value should not be transformed into pointer to pointers. Table 6 shows an example that correctly handle this case.

Scala	<pre>def foo(x: Int) = {
	  def bar(y: Int) = x * y
	  bar(2)
	}</pre>

C99	<pre>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);
	}</pre>

Table 5: Example of simply nested functions

3.4 Statements

GenC provides support for converting basic operators, `if`- and `while`-constructs to C99 as described next by ensuring a consistent execution of instructions as well as converting traditional functional form to imperative style.

3.4.1 Operators

The support for operators is shown in Table 7. Note that both unary and binary minus operators are supported – but Scala's `>>>` is not – and that the effect of operators is the same in both languages. Additionally, the precedence of

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

C99	```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);
	}```

Table 6: Example of deeply nested functions

operators, even though roughly equivalent, is guaranteed in the generated code by wrapping every sub-expressions in parentheses conformally to Scala operator priority hierarchy.

Category	Operators
Boolean operators	`&& || ! != ==`
Comparison operators over integers	`< <= == != >= >`
Arithmetic operators over integers	`+ - * / %`
Bitwise operators over integers	`& | ^ - << >>`

Table 7: Supported operators

3.4.2 if-Construct

One major difference between conditional branching in Scala vs C is the ability to assign a variable to the value generated by an **if**-construct in Scala. *GenC* handles these situations by transforming values into variables and assigns the appropriate value inside the *then* or *else* branches of the **if**-statement in C. Similarly, when an **if**-construct is used as the last instruction of a function, Scala will return it. In C99, instead of creating a temporary variable and then returning it, *GenC* injects a **return** statement in both branches of the conditional

Scala	`def abs(x: Int) = if (x >= 0) x else -x`
	
	`def fun(b: Boolean) {`
	`  val x = if (b) 42 else 58`
	`  // ...`
	`}`

C99	`int32_t abs0(int32_t const x0) {`
	`  if ((x0 >= 0)) { return x0; }`
	`  else { return (-x0); }`
	`}`
	
	`void fun0(bool const b0) {`
	`  int32_t x8; // no const here`
	`  if (b0) { x8 = 42; }`
	`  else { x8 = 58; }`
	`  // ...`
	`}`

Table 8: Examples of `if`-construct conversions

branching. Table 8 shows examples of this kind of conversions.

3.4.3 `while`-Construct

Basic **`while`**-construct are supported rather straightforwardly by *GenC* as the meaning of this keyword is the same in both languages. Table 9 illustrates this. However, when nested instructions are put inside the loop condition, the conversion becomes slightly more tricky as exposed in Section 3.4.4.

3.4.4 Execution Order Normalisation

On the one hand, we have Scala, based on JVM runtime, that enforces a strict order of instruction execution at runtime. Generally speaking, Scala executes statement from left to right. On the other hand, we have the C standard that specifies a rather flexible order of execution. Compiler manufacturers can choose to re-order instructions in order to optimise code. The C99 standard defines the behaviour of program execution through Sequence Points[5] and Undefined Behaviour. As a direct consequence, expressions are often not executed from left to right but in a implementation-specific order. Depending on the compiler and target architecture, the same C code can present different behaviours. We therefore have to be careful when converting instructions from Scala to C in

[5]Refers to Section 5.1.2.3 and Annex C of the C99 standard for the complete definition of sequence points.

Scala	<pre>def sum(a: Array[Int]): Int = {
	  var i = 0
	  var acc = 0
	  while (i < a.length) {
	    acc = acc + a(i)
	    i = i + 1
	  }
	  acc
	}</pre>

C99	<pre>int32_t sum0(__leon_array_int32_t_t const a0) {
	  int32_t i9 = 0;
	  int32_t acc1 = 0;
	  while ((i9 < a0.length)) {
	    acc1 = (acc1 + a0.data[i9]);
	    i9 = (i9 + 1);
	  }
	  return acc1;
	}</pre>

Table 9: Example of `while`-construct conversions

order to prevent operations with side-effect to work differently in both languages but also with all C compilers and target architectures.

There are specific areas of code where this issue is fundamentally important. For example in Table 10, when invoking the function `foo` with its two arguments, both parameters need to be evaluated from left to right. *GenC* makes sure that the sequence points match the behaviour of the original program by creating intermediary variables wherever needed.

In addition to the execution normalisation of function parameters, *GenC* extracts nested statements in operands and branching conditions, while preserving short-circuiting in boolean operations but regardless of operator precedence. Tables 11 and 12 illustrate those two scenarios. In the first example, sub-expressions can simply be lifted outside their respective block by transforming the left-to-right ordering into top-to-bottom. In the second example, boolean conjunction and disjunction operators are replaced by their equivalent **`if`**-statement. Additionally, the unnecessary last term was removed from the C translation, thanks to the short circuiting rules.

A slightly more complex normalisation example is shown in Table 13 where a **`while`**-statement needs to be expressed in terms of an infinite loop with a break condition in order to accommodate for the boolean condition having some side-effect and therefore requiring to be re-evaluated at each loop iteration. Furthermore, since the second conditional operand has no side-effect, the resulting decision tree is compressed into two **`if`**-statements instead of three.

Scala	<pre>def example = {
	  var counter = 0;
	  def get() = {
	    counter = counter + 1
	    counter
	  }
	  def foo(x: Int, y: Int) = {
	    if (x < y) true
	    else false
	  }
	
	  foo(get(), get())
	}</pre>

C99	<pre>int32_t get2(int32_t* counter0) {
	  (*counter0) = ((*counter0) + 1);
	  return (*counter0);
	}
	
	bool foo0(int32_t* counter0, int32_t const x7, int32_t const y6) {
	  if ((x7 < y6)) { return true; }
	  else { return false; }
	}
	
	bool example0(void) {
	  int32_t counter0 = 0;
	  int32_t const __leon_normexec0 = get2((&counter0));
	  int32_t const __leon_normexec1 = get2((&counter0));
	  return foo0((&counter0), __leon_normexec0, __leon_normexec1);
	}</pre>

Table 10: Example of execution normalisation of function call in C99

Scala	`def example(j: Int) = {`
	`  var c = 0`
	`  val x = { c = c + 3; j } +`
	`          { c = c + 1; j } * { c = c * 2; j }`
	`  c == 8`
	`}`

C99	`bool example0(int32_t const j0) {`
	`  int32_t c2 = 0;`
	`  c2 = (c2 + 3);`
	`  c2 = (c2 + 1);`
	`  c2 = (c2 * 2);`
	`  int32_t const x7 = (j0 + (j0 * j0));`
	`  return (c2 == 8);`
	`}`

Table 11: Example of execution normalisation for operands in C99

Scala	`def example(b: Boolean) = {`
	`  val f = b && !b // == false`
	`  var c = 0`
	`  val x = f || { c = 1; true } || { c = 2; false }`
	`  c == 1`
	`}`

C99	`bool example0(bool const b0) {`
	`  bool const f26 = (b0 && (!b0));`
	`  int32_t c2 = 0;`
	`  bool x7;`
	`  if (f26) { x7 = true; }`
	`  else {`
	`    c2 = 1;`
	`    x7 = true;`
	`  }`
	`  return (c2 == 1);`
	`}`

Table 12: Example of execution normalisation with short-circuiting in C99

Scala	<pre>def example(b: Boolean) = {
	  var i = 10
	  var c = 0
	
	  val f = b && !b // == false
	  val t = b || !b // == true
	
	  // The following condition is executed 11 times,
	  // and only during the last execution is the last
	  // operand evaluated
	  while ({ c = c + 1; t } && i > 0 || { c = c * 2; f }) {
	    i = i - 1
	  }
	
	  i == 0 && c == 22
	}</pre>

C99	<pre>bool example0(bool const b0) {
	  int32_t i9 = 10;
	  int32_t c2 = 0;
	  bool const f26 = (b0 && (!b0));
	  bool const t8 = (b0 || (!b0));
	
	  while (true) {
	    c2 = (c2 + 1);
	    if ((t8 && (i9 > 0))) { /* empty */ }
	    else {
	      c2 = (c2 * 2);
	      if (f26) { /* empty */ }
	      else { break; }
	    }
	
	    i9 = (i9 - 1);
	  }
	
	  return ((i9 == 0) && (c2 == 22));
	}</pre>

Table 13: Example of execution normalisation of `while`-statement in C99

3.4.5 Ignored Features

The following Scala statements are ignored by *GenC* as they should have no side-effect except decreasing the runtime performance: `require` (pre-conditions), `ensuring` (post-conditions) and `assert`. It is expected from the user that those statements were verified using Leon verification mechanisms.

4 Case Study

To assess the quality of this project, a case study showcasing basic image processing was implemented in Scala and verified using Leon and then converted to C99 using its *GenC* phase. Appendices A and B report the Scala source code and its analogous C99 code. This example highlights the current capability of Leon to generate safe C code, but also shows where it should be improved.

The code can be verified using:

```
leon --xlang --solvers=smt-z3,ground IntegralColor.scala
```

and converted to C99 with:

```
leon --xlang --genc --o=IntegralColor.c IntegralColor.scala
```

The first limitation is the absence of support for floating point types such as **`double`** in C. This limitation is due to the complexity involved when proving properties about such variables. Therefore, one has to convert operations on real numbers into integer operations, which can substantially complexify algorithms such as in image processing algorithms.

A second serious limitation is the impossibility to mutate array passed as parameter to functions or to return arrays from functions. This is, respectively, due to aliasing of variables, which makes it harder to prove some properties, and to the simple memory model currently implemented by *GenC*. Hence, the modularity of the code is significantly impacted: for example, it is not possible to define a function that converts an arbitrary RGB-image into greyscale using arrays. Instead, one has to duplicate code in order to apply, say smoothing, to any image.

However, we were able to prove the correctness of this case study – which exhibits close to every supported concepts – and the produced C99 code could be successfully compiled using Clang. Furthermore, when executed, the generated program shows the appropriate behaviour.

5 Future Development

In this initial version of *GenC*, we were able to support not only basic types such as boolean or 32-bit integer, but also generate appropriate code to define and use tuple of arbitrary types, immutable case classes and stack-allocated arrays of either fixed size or runtime size. The distinction between **`val`** and **`var`** for those types and the idiom of mutability were applied to C program through constness. Additionally, overloads of functions and nested functions were properly converted to the namespace-less C language. Finally, imperative code instructions such as `if`- or `while`-statements, as well as usual operators on the supported basic types, were translated appropriately, enforcing a consistent execution order of expressions for both languages.

The next steps of development for this module of Leon are multiple. On a general level, Leon and *GenC* could be extended to support verifying some properties about floating point types, or to ensure that no overflow occurs, but also by supporting additional integral types such as 16-bit or 64-bit integers.

GenC could also be improved by providing a more complex memory model that supports heap-allocation. Since this feature might not be usable on some devices, it would probably be safer to make it opt-in through a command line flag. This advanced model would help writing modular code such as discussed in Section 4.

An important part of code written in Scala is based on functional features. One of the major features being higher-order functions, supporting them in *GenC* would significantly improve the range of program that could be safely converted to C99.

Regarding functions, currently *GenC* considers every function to have side-effect. Execution normalisation as presented in Section 3.4.4 could benefit from detecting which function can have side-effect for the current statement to generate code that not only would be easier to read but also let C compilers perform optimisations at will. Moreover, while not always necessarily useful in practice since C compiler are allowed to remove unused parameters to optimise code, the parameters of extracted nested functions could be reduced to the minimal set of variables that are accessed.

Finally, some secondary details could be improved such as the pretty printer to improve code readability: extra parentheses could be removed and identifiers could be kept as they are in the input source code where there is no ambiguity due to name clashing in a namespace-less language such as C. We however believe that improving further the pretty printer regarding the formatting of the produced code would not be useful since some advanced tools are already independently developed and widely used.

A Case Study: IntegralColor.scala

```scala
import leon.lang._

object IntegralColor {

  def isValidComponent(x: Int) = x >= 0 && x <= 255

  def getRed(rgb: Int): Int = {
    (rgb & 0x00FF0000) >> 16
  } ensuring isValidComponent _

  def getGreen(rgb: Int): Int = {
    (rgb & 0x0000FF00) >> 8
  } ensuring isValidComponent _

  def getBlue(rgb: Int): Int = {
    rgb & 0x000000FF
  } ensuring isValidComponent _

  def getGray(rgb: Int): Int = {
    (getRed(rgb) + getGreen(rgb) + getBlue(rgb)) / 3
  } ensuring isValidComponent _

  def testColorSinglePixel: Boolean = {
    val color = 0x20C0FF

    32 == getRed(color) && 192 == getGreen(color) &&
    255 == getBlue(color) && 159 == getGray(color)
  }.holds

  def matches(value: Array[Int], expected: Array[Int]): Boolean = {
    require(value.length == expected.length)

    var test = true
    var idx = 0
    (while (idx < value.length) {
      test = test && value(idx) == expected(idx)
      idx = idx + 1
    }) invariant { idx >= 0 && idx <= value.length }

    test
  }

  def testColorWholeImage: Boolean = {
    val WIDTH  = 2
    val HEIGHT = 2

    val source   = Array(0x20c0ff, 0x123456, 0xffffff, 0x000000)
```

```
    val expected = Array(159, 52, 255, 0) // gray convertion
    val gray     = Array.fill(4)(0)

    // NOTE: Cannot define a toGray function as XLang
    // doesn't allow mutating arguments and GenC doesn't
    // allow returning arrays

    var idx = 0
    (while (idx < WIDTH * HEIGHT) {
      gray(idx) = getGray(source(idx))
      idx = idx + 1
    }) invariant {
      idx >= 0 && idx <= WIDTH * HEIGHT &&
      gray.length == WIDTH * HEIGHT
    }
    // NB: the last invariant is very important;
    // without it the verification times out

    matches(gray, expected)
  }.holds

  // Only for square kernels
  case class Kernel(size: Int, buffer: Array[Int])

  def isKernelValid(kernel: Kernel): Boolean =
    kernel.size > 0 && kernel.size < 1000 && kernel.size % 2 == 1 &&
    kernel.buffer.length == kernel.size * kernel.size

  def applyFilter(gray: Array[Int], size: Int,
                  idx: Int, kernel: Kernel): Int = {
    require(size > 0 && size < 1000 &&
            gray.length == size * size &&
            idx >= 0 && idx < gray.length &&
            isKernelValid(kernel))

    def up(x: Int): Int = {
      if (x < 0) 0 else x
    } ensuring { _ >= 0 }

    def down(x: Int): Int = {
      if (x >= size) size - 1 else x
    } ensuring { _ < size }

    def fix(x: Int): Int = {
      down(up(x))
    } ensuring { res => res >= 0 && res < size }

    def at(row: Int, col: Int): Int = {
      val r = fix(row)
      val c = fix(col)
```

```

      gray(r * size + c)
    }

    val mid = kernel.size / 2

    val i = idx / size
    val j = idx % size

    var res = 0
    var p = -mid
    (while (p <= mid) {
      var q = -mid

      (while (q <= mid) {
        val krow = p + mid
        val kcol = q + mid

        assert(krow >= 0 && krow < kernel.size)
        assert(kcol >= 0 && kcol < kernel.size)

        val kidx = krow * kernel.size + kcol

        res += at(i + p, j + q) * kernel.buffer(kidx)

        q = q + 1
      }) invariant { q >= -mid && q <= mid + 1 }

      p = p + 1
    }) invariant { p >= -mid && p <= mid + 1 }

    res
  }

  def testFilterConvolutionSmooth: Boolean = {
    val gray = Array(127, 255, 51, 0)
    val expected = Array(124, 158, 76, 73)
    val size = 2 // grey is size x size

    // NOTE: Cannot define a ‘smoothed‘ function as XLang
    // doesn’t allow mutating arguments and GenC doesn’t
    // allow returning arrays

    val kernel = Kernel(3, Array(1, 1, 1,
                                 1, 2, 1,
                                 1, 1, 1))

    val smoothed = Array.fill(gray.length)(0)
    assert(smoothed.length == expected.length)

```

```
    var idx = 0;
    (while (idx < smoothed.length) {
      smoothed(idx) = applyFilter(gray, size, idx, kernel) / 10
      idx = idx + 1
    }) invariant {
      idx >= 0 && idx <= smoothed.length &&
      smoothed.length == gray.length
    }

    matches(smoothed, expected)
  }.holds

  def main: Int = {
    if (testColorSinglePixel &&
        testColorWholeImage &&
        testFilterConvolutionSmooth) 0
    else 1
  } ensuring { _ == 0 }

}
```

B Case Study: Generated IntegralColor.c

```c
/* ----------------------------------- includes ----- */

#include <assert.h>
#include <stdbool.h>
#include <stdint.h>

/* --------------------- data type declarations ----- */

struct __leon_array_int32_t_t;
struct Kernel0;

/* ---------------------- data type definitions ----- */

typedef struct __leon_array_int32_t_t {
  int32_t* data;
  int32_t length;
} __leon_array_int32_t_t;

typedef struct Kernel0 {
  int32_t const size0;
  __leon_array_int32_t_t const buffer0;
} Kernel0;

/* ---------------------- function declarations ----- */

bool
isValidComponent0(int32_t const x0);

int32_t
getRed0(int32_t const rgb0);

int32_t
getGreen0(int32_t const rgb1);

int32_t
getBlue0(int32_t const rgb2);

int32_t
getGray0(int32_t const rgb3);

bool
testColorSinglePixel0(void);

bool
matches0(__leon_array_int32_t_t const value0, __leon_array_int32_t_t const expected0);

```

```
bool
testColorWholeImage0(void);

bool
isKernelValid0(Kernel0 const kernel0);

int32_t
up0(__leon_array_int32_t_t const* gray0, int32_t const* size1, int32_t const* idx0, Kernel0 const* kernel1, int32_t const x12);

int32_t
down0(__leon_array_int32_t_t const* gray0, int32_t const* size1, int32_t const* idx0, Kernel0 const* kernel1, int32_t const x13);

int32_t
fix0(__leon_array_int32_t_t const* gray0, int32_t const* size1, int32_t const* idx0, Kernel0 const* kernel1, int32_t const x14);

int32_t
at0(__leon_array_int32_t_t const* gray0, int32_t const* size1, int32_t const* idx0, Kernel0 const* kernel1, int32_t const row0, int32_t const col0);

int32_t
applyFilter0(__leon_array_int32_t_t const gray0, int32_t const size1, int32_t const idx0, Kernel0 const kernel1);

bool
testFilterConvolutionSmooth0(void);

int32_t
main(void);

/* ------------------------ function definitions ----- */

bool
isValidComponent0(int32_t const x0)
{
  return ((x0 >= 0) && (x0 <= 255));
}

int32_t
getRed0(int32_t const rgb0)
{
  return ((rgb0 & 16711680) >> 16);
}

int32_t
getGreen0(int32_t const rgb1)
{
```

```
  return ((rgb1 & 65280) >> 8);
}

int32_t
getBlue0(int32_t const rgb2)
{
  return (rgb2 & 255);
}

int32_t
getGray0(int32_t const rgb3)
{
  int32_t const __leon_normexec0 = getRed0(rgb3);
  int32_t const __leon_normexec1 = getGreen0(rgb3);
  int32_t const __leon_normexec2 = getBlue0(rgb3);
  return (((__leon_normexec0 + __leon_normexec1) + __leon_normexec2) /
    ↪ 3);
}

bool
testColorSinglePixel0(void)
{
  int32_t const color0 = 2146559;
  int32_t const __leon_normexec3 = getRed0(color0);
  if ((32 == __leon_normexec3))
  {
    int32_t const __leon_normexec4 = getGreen0(color0);
    if ((192 == __leon_normexec4))
    {
      int32_t const __leon_normexec5 = getBlue0(color0);
      if ((255 == __leon_normexec5))
      {
        int32_t const __leon_normexec6 = getGray0(color0);
        return (159 == __leon_normexec6);
      }
      else
      {
        return false;
      }

    }
    else
    {
      return false;
    }

  }
  else
  {
    return false;
```

```
  }

}

bool
matches0(__leon_array_int32_t_t const value0, __leon_array_int32_t_t const expected0)
{
  bool test0 = true;
  int32_t idx1 = 0;
  while ((idx1 < value0.length))
  {
    test0 = (test0 && (value0.data[idx1] == expected0.data[idx1]));
    idx1 = (idx1 + 1);
  }

  return test0;
}

bool
testColorWholeImage0(void)
{
  int32_t const WIDTH0 = 2;
  int32_t const HEIGHT0 = 2;
  int32_t __leon_buffer12[4] = { 2146559, 1193046, 16777215, 0 };
  __leon_array_int32_t_t const source0 = { .length = 4, .data = __leon_buffer12 };

  int32_t __leon_buffer13[4] = { 159, 52, 255, 0 };
  __leon_array_int32_t_t const expected1 = { .length = 4, .data = __leon_buffer13 };

  int32_t __leon_buffer14[4] = { 0, 0, 0, 0 };
  __leon_array_int32_t_t const gray1 = { .length = 4, .data = __leon_buffer14 };

  int32_t idx2 = 0;
  while ((idx2 < (WIDTH0 * HEIGHT0)))
  {
    int32_t const __leon_normexec7 = getGray0(source0.data[idx2]);
    gray1.data[idx2] = __leon_normexec7;
    idx2 = (idx2 + 1);
  }

  return matches0(gray1, expected1);
}

bool
isKernelValid0(Kernel0 const kernel0)
{
```

```
  return ((kernel0.size0 > 0) && (kernel0.size0 < 1000) && ((kernel0.size0 % 2) == 1) && (kernel0.buffer0.length == (kernel0.size0 * kernel0.size0)));
}

int32_t
up0(__leon_array_int32_t_t const* gray0, int32_t const* size1, int32_t const* idx0, Kernel0 const* kernel1, int32_t const x12)
{
  if ((x12 < 0))
  {
    return 0;
  }
  else
  {
    return x12;
  }

}

int32_t
down0(__leon_array_int32_t_t const* gray0, int32_t const* size1, int32_t const* idx0, Kernel0 const* kernel1, int32_t const x13)
{
  if ((x13 >= (*size1)))
  {
    return ((*size1) - 1);
  }
  else
  {
    return x13;
  }

}

int32_t
fix0(__leon_array_int32_t_t const* gray0, int32_t const* size1, int32_t const* idx0, Kernel0 const* kernel1, int32_t const x14)
{
  int32_t const __leon_normexec8 = up0(gray0, size1, idx0, kernel1, x14);
  return down0(gray0, size1, idx0, kernel1, __leon_normexec8);
}

int32_t
at0(__leon_array_int32_t_t const* gray0, int32_t const* size1, int32_t const* idx0, Kernel0 const* kernel1, int32_t const row0, int32_t const col0)
{
  int32_t const r3 = fix0(gray0, size1, idx0, kernel1, row0);
```

```
  int32_t const c2 = fix0(gray0, size1, idx0, kernel1, col0);
  return (*gray0).data[((r3 * (*size1)) + c2)];
}

int32_t
applyFilter0(__leon_array_int32_t_t const gray0, int32_t const size1,
    ↪int32_t const idx0, Kernel0 const kernel1)
{
  int32_t const mid0 = (kernel1.size0 / 2);
  int32_t const i9 = (idx0 / size1);
  int32_t const j0 = (idx0 % size1);
  int32_t res2 = 0;
  int32_t p17 = (-mid0);
  while ((p17 <= mid0))
  {
    int32_t q0 = (-mid0);
    while ((q0 <= mid0))
    {
      int32_t const krow0 = (p17 + mid0);
      int32_t const kcol0 = (q0 + mid0);
      int32_t const kidx0 = ((krow0 * kernel1.size0) + kcol0);
      int32_t const __leon_normexec9 = at0((&gray0), (&size1), (&idx0)
    ↪, (&kernel1), (i9 + p17), (j0 + q0));
      res2 = (res2 + (__leon_normexec9 * kernel1.buffer0.data[kidx0]))
    ↪;
      q0 = (q0 + 1);
    }

    p17 = (p17 + 1);
  }

  return res2;
}

bool
testFilterConvolutionSmooth0(void)
{
  int32_t __leon_buffer15[4] = { 127, 255, 51, 0 };
  __leon_array_int32_t_t const gray2 = { .length = 4, .data =
    ↪__leon_buffer15 };

  int32_t __leon_buffer16[4] = { 124, 158, 76, 73 };
  __leon_array_int32_t_t const expected2 = { .length = 4, .data =
    ↪__leon_buffer16 };

  int32_t const size4 = 2;
  int32_t __leon_buffer17[9] = { 1, 1, 1, 1, 2, 1, 1, 1, 1 };
  __leon_array_int32_t_t const __leon_normexec10 = { .length = 9, .
    ↪data = __leon_buffer17 };

```

```
  Kernel0 const kernel2 = (Kernel0) { .size0 = 3, .buffer0 = __leon_normexec10 };
  int32_t __leon_vla_buffer18[gray2.length];
  for (int32_t __leon_i19 = 0; __leon_i19 < gray2.length; ++__leon_i19) {
    __leon_vla_buffer18[__leon_i19] = 0;
  }
  __leon_array_int32_t_t const smoothed0 = { .length = gray2.length, .data = __leon_vla_buffer18 };

  int32_t idx3 = 0;
  while ((idx3 < smoothed0.length))
  {
    int32_t const __leon_normexec11 = applyFilter0(gray2, size4, idx3, kernel2);
    smoothed0.data[idx3] = (__leon_normexec11 / 10);
    idx3 = (idx3 + 1);
  }

  return matches0(smoothed0, expected2);
}

int32_t
main(void)
{
  if (testColorSinglePixel0())
  {
    if (testColorWholeImage0())
    {
      if (testFilterConvolutionSmooth0())
      {
        return 0;
      }
      else
      {
        return 1;
      }

    }
    else
    {
      return 1;
    }

  }
  else
  {
    return 1;
  }
}
```