Extending Safe C Support In Leon

Marco Antognini

January 2017

Abstract

As hardware designs get more robust and efficient, software can solve a wider range of challenges, each one more advanced than the previous one. The direct consequence is that software complexity grows continuously. Despite being used more frequently in development processes, traditional testing frameworks are not enough to avoid flaws in programs. Moreover, many company concerned by the security of their applications, especially in mission critical industries, spend a considerable effort and amount of money to write robust software certified by solid guidelines and standards. For embedded systems, MISRA C is a certification process commonly used that provides directive to write portable and maintainable C code while avoiding many pitfalls of the language.

Leon takes an orthogonal approach by providing tools for formal verification of programs written in a subset of Scala. In a previous work we introduced *GenC*, a module for Leon that converts Scala applications into equivalent and safe C99 programs, allowing developers to leverage high-level feature of Scala to implement robust applications even for embedded systems. The aim of this project is to augment Leon and *GenC* in order to support a larger fragment of Scala that includes inheritance, generic programming, additional numeric types, pattern matching and more. Additionally, we closely analyse the MISRA Guidelines and shape the generated C code to work towards compliance with its rules while taking advantage of the verification capabilities of Leon to automatically detect a variety of bugs. We discuss these improvements and new features by implementing the famous LZW compression/decompression algorithm.

Master Thesis Project under the supervision of
Prof. Viktor Kuncak & Régis Blanc
Lab for Automated Reasoning and Analysis LARA - EPFL

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Contents

1 Introduction

Software become significantly more complex, solving every day new challenges, partly thanks to the decrease in price of hardware which is constantly improved, making computers more and more robust, quicker and more energy efficient. Constant breakthroughs enabled us to revolutionise how people benefits from their personal computers, or electronic devices in a general sense, to ends unforeseeable in some cases no longer than a decade ago: from the information now instantaneously available at our finger tips provided by satellites orbiting our planet in an environment ranging from extremely cold to scorching temperature with intense radiations, through remote devices performing advanced scientific experiments at an incredibly huge distance from us communicating using channels resilient to immense delay and numerous data errors, to surgical and medical tools helping to understand and to fight devastating diseases, for example using brain or heart implements to improve patients' life.

Future technologies can be expected to follow a similar evolution, making obvious that systems complexity is predicted to not decrease at all. Many companies already spend a considerable amount of money to assert their software is exempt of life threatening bugs through various certifications or to keep their data safe and available in a matter of jiffies. Hence, being able to evaluate the quality of software, detecting issues before they arise, in a reliable and simple fashion can significantly boost performance of Quality Assurance (QA) processes.

Development strategies were developed to actively reinforce software. They can involve continuous integration servers to constantly check for defects by means of regression or unit tests and powerful static analysers. Or they can be based on general design techniques, such as employing design-by-contract programming or runtime validations and exception mechanisms.

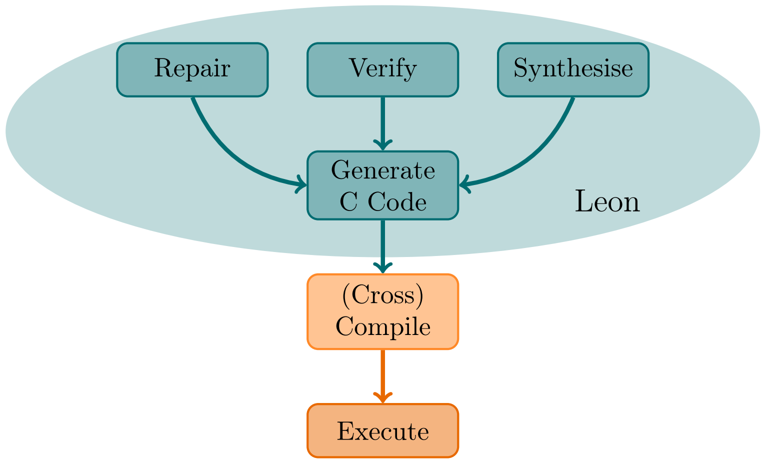

Figure 1: Development Process

For programs written in Scala, Leon [1] is a powerful instrument that works at solving this general issue of quality 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, native languages often taking their roots in the C language.

For these reasons we have previously introduced the *GenC* module of Leon [2] to transpile Scala programs into a safe and equivalent C99 pieces of software. *GenC* allows programmers to leverage the high-level features of the Scala programming language and reduce the development cost of low-level software while avoiding using error-prone languages. Moreover it helps developers avoid some of the most commonly reported bugs (e.g. buffer overflows or arithmetic errors).

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. Then the program can be converted into an equivalent C99 code through the *GenC* phase. The produced code can then be compiled using any standard-compliant C99 compiler – for example Clang [3], GCC [4] or formally verified compilers like CompCert [5] [6] – 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.

The present work adds to *GenC* a better support for object oriented or generic programming, a wider range of supported primitive types, an improved handling of arithmetic expressions and an interfacing system to integrate existing libraries, just to name a few. Together these improvements augment the range of the supported fragments of Scala. We also introduce a new memory model based on our analyse of recent improvements made to the *xlang* module [7] [8] regarding mutability. One singular feature of this model is that it is solely based on stack allocation.

Furthermore, we studied security processes used for development of applications targeted at embedded devices, such as the ones used in the automotive industry. This work resulted in a re-engineering of *GenC* to work toward compliance with the MISRA Guidelines, which is intended to bring in robustness and reliability to the software, and to empower users with more static analysis tools. We discuss in this document how verification can help seriously reduce the amount of bugs in programs and report on how the Leon ecosystem manages to produce safe code free of undefined behaviour and under which assumptions.

We also present two implementation of the famous LZW [9] compression and decompression algorithm, illustrating how *GenC* can successfully be used for real-life applications.

We structure this report as follow. First we finish this section by reviewing some related works. In Section 2 we highlight the most common kinds of bugs and what are the main challenges for embedded development. Then, we present new features we developed for Leon itself in Section 3. We introduce the memory model on which the generated C99 code is relying on in Section 4 and further discuss specific details of the translation from Scala down to C in Sections 5 and 6. Next, we specify how external libraries can be integrated into the translation process in Section 7. In Section 8 we present the MISRA Guidelines, discuss its importance and explain how Leon and *GenC* work toward compliance. Then, we showcase in Section 9 the abilities of *GenC* by implementing the LZW algorithm. We summarise this report in Section 10 and in the final Section 11 we introduce how Leon and *GenC* can be further improved to help developing safe applications.

Related Works

Because *GenC* spans several fields it is relevant to identify whether some existing project proposes to transpile a high-level language into embedded-friendly equivalent code, what kind of verification systems exist or how bugs are tracked down in compilers.

First, we note that *GenC* is not the first attempt to translate a high-level language down to C. There exists, for example, the now discontinued java2c [10] project which claims to transpile a limited fragment of Java to C. However, contrary to java2c, *GenC* is built into a wider ecosystem that make such translation even more meaningful thanks to the different modules of Leon.

Frama-C [11] is somewhat similar to the static analysis modules of Leon. Both offer to verify without actually running programs whether some properties hold or not. But while Leon works with programs written in Scala, Frama-C analyses software in C. Which means that higher level concepts such as inheritance or generic types cannot be used to design complex programs, therefore limiting developers to basic tools. That being said, adding assertions and properties written in ANSI/ISO C Specification Language (ACSL) [12], which interface nicely with C since it can be embedded in standard comment block, to the generated C99 code based on the original verification conditions the user wrote in Scala would be beneficial to help prove the new, low-level program still respects its original specification.

Finally, to design any program bigger than a few assembly instructions one will need to use a compiler for whatever language is being used. However, those specific programs are no different from the one most developers write: they can contain bugs. And because those defects could impact any application, regardless of its domain, and therefore result in disastrous consequences. In that respect, *GenC* indirectly relies on project such as Csmith [13] to improve the quality of compilers, especially the C ones, in order to test its ability to produce valid C. While *GenC* currently only relies on classic regression tests, it could be interesting to use a similar generator of random test-case than the one used by Csmith to further assess its ability to transpile Scala code.

2 Motivation

We begin this report by exposing how frequent and dangerous bugs can be through diverse examples and issue databases. The second part of this section further details which additional challenges need to be addressed when working with embedded devises, introduce the MISRA Guidelines used by the industry to certify the quality of programs and how *GenC* fit in this certification process.

2.1 A Long History of Bugs

There are some well-known bugs in the history of computer science that caused major monetary loss to companies, and in a few cases even caused physical injuries or death. But there also exist countless bugs that are more moderate but still threaten the security of our information systems and everyday computers. In some specific cases bugs evolved into features, and in particularly odd cases vulnerabilities were abused to patch the software itself. Here are a few examples:

Killer Bug, Therac-25 The Therac-25, which was used in the 80's for radiation therapy, caused 6 known accidents involving massive overdoses resulting in severe injuries and deaths. An integer used to track error count overflowed, causing the program to re-enter a valid state for operation [14].

To fight against such bugs we instrumented Leon to check for potential integer overflow without running the software.

(s)elf-exploitation From time to time, a bug happen to be situated in the right position to workaround another limitation. J. Garret explained in [15] how a buffer overflow was used to patch a game in the most unexpected way. He also acknowledges that this wouldn't have been needed were the software including a remote patch procedure.

Out-of-Bounds Access & Buffer Overflow Some of the most frequent kinds of bugs present in our software are overflows; for example Apple Security Update HT207275 [16] patches several bugs of which roughly 20% were *simple* buffer overflows and out-of-bounds accesses.

Those are typically bugs that can be avoided using Leon and *GenC*: the verification procedure highlight place in Scala programs where overflows can occur, with an example of values that triggers the bug at runtime.

According to the widely adopted *Common Vulnerabilities and Exposures* (CVE) database [17], which describes itself as *The Standard for Information Security Vulnerability Names* and has over 80,000 registered tickets, buffer overflows account for approximately 10% of the reported bugs and integer overflows for roughly 1.6%.

The *CVE Details* website [18] reports that overflows, in its general meaning, is third in a vulnerabilities-by-type ranking. One might expect that overflows are sufficiently studies and known by developers that their presence in our modern software should not contain many such bugs. However, as depicted by Figure 2, the fact is that the general trend is not toward a reduction of the number of discovered bugs in this category. After a closer reading of the CVE entries it appears that many vulnerabilities categorised as *Denial of Service* or *Execute Code* are also tagged as *Overflow.*

It follows that people need clever tools, such as Leon, to detect and eradicate those omnipresent issues.

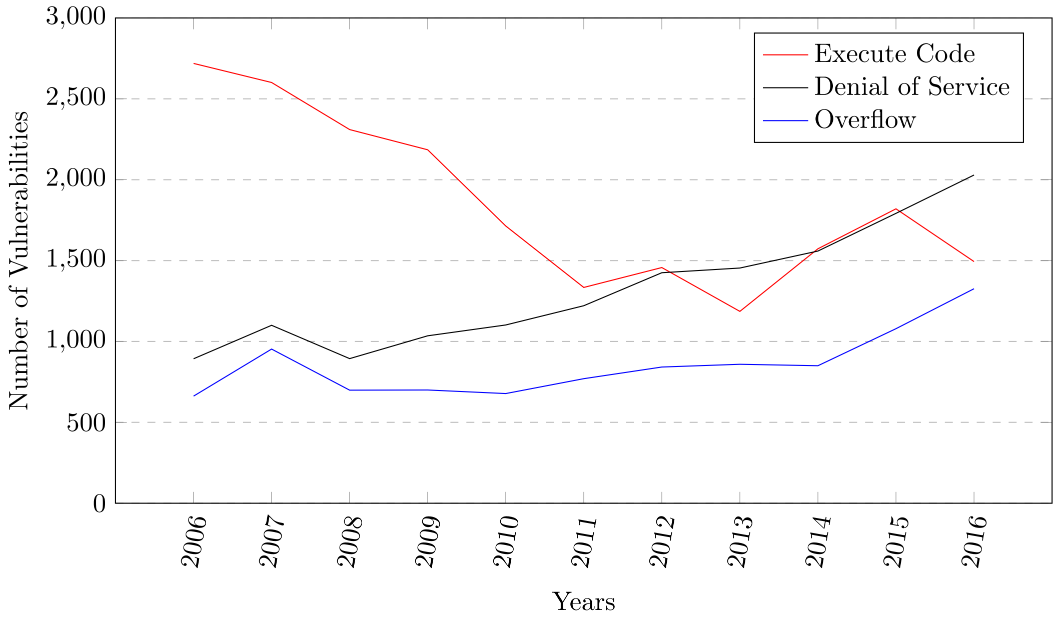

Figure 2: The 3 most frequent categories of bugs in the last 10 years. Source: [18]

2.2 Challenges for Embedded Devices

Seen the range of bugs and their frequencies and gravity, it appears clear that developers needs good tools in order to write decently safe software. Embedded devices add to this context the challenge of producing safe **and** efficient code with a limited amount of resources, such as computational speed, memory capacity and energy consumption. C is commonly chosen to develop programs for such restrictive environments. The reason for this choice is twofold. Firstly, in contrast to Java or Python it is a native language and therefore is more suited for such tiny runtime environments. Secondly, despite the work of many C++ developers C remains the preferred language in practice with an adoption of 60% [19], certainly for historical reasons but also because most suspect that *high-level* language features such as meta-programming or object oriented design are too greedy.

However, it is well-known that C is not an easy language and that many pitfalls make developers' life hard. Furthermore, if one wants to write portable software, one has to work through numerous language elements that are either implementation-defined or described as undefined behaviour by the C Standard. To get an idea of the difficulty this represents, the C99 Standard [20] in its Annex J holds a bullet-point list of such undefined elements that is 13 pages long.

The following example is one of the shortest function that can trigger undefined behaviour, illustrating the extend to which the language is implementation-defined:

```c
int foobar(int x) { return -x; } // undefined when x == INT_MIN
```

It is not obvious in this snippet what is the ranged covered by **int**: the C99 standard specifies a minimal size for **int** but leaves the implementation define its size as well as its memory representation, with some constraints. It could theoretically be defined as a 19-bit integer using 1's complement representation.

On the other end of the spectrum, the Java programming language and its virtual machine, and therefore Scala, precisely define the range of **int** (or **Int** in Scala) and its memory representation as a 32-bit integer using the now standard 2's complement representation. Moreover, the behaviour of the unary minus operator is properly defined by the Java Standard [21] on the whole $[-2^{31}, 2^{31} - 1]$ range using wrap around semantic for overflow, meaning that $-(-2^{31}) \equiv -2^{31}$.

On top of the C standard, many companies, especially in the car, aeronautic or medical industries, adopt additional guidelines to improve the quality of their software. They tend to be very strict and even forbid some of the language features deemed unsafe or unpredictable. One of the most used set of rules is *MISRA C: Guidelines for the use of the C language in critical systems* [22].

Among many things, the MISRA Guidelines forbid any kind of dynamic memory allocation, leaving only static allocation available to the developers and therefore significantly limiting program designs; the rationale being that heap allocation is hard to predict not only in terms of available memory space but also, and this is equally important for real-time application, in term of responsiveness [23] and is therefore inadequate for most embedded systems.

One of the direct consequences of this memory allocation model is that a traditional `List` in Scala cannot be implemented trivially in C. Listings 1 and 2 illustrate this.

```scala
abstract class List
case class Cons(x: Int, tail: List) extends List
case class Nil() extends List
```

Listing 1: List of integers in Scala.

With *GenC*, developers can use Leon produce low level C99 code as safe as the verified Scala program given as input without having to deal with portability issues. By using a language like Scala, programmers can take advantage of the strong type system to avoid illegal type conversions that emerge from the needs of **void**∗ in C due to the absence of generic types, but also benefit from the well-defined nature of the language.

Additionally, we believe it is possible to write safe **and** efficient code, as we hint with the example of Listing 3. We can write two different `append` functions for a list of fixed maximum capacity. One with a wide contract that returns an error when the list is already full and the other one with a narrow contract that appends the element without checking the capacity but should be called exclusively on non-full lists. With Leon, we can ensure that calls to the optimised version are valid, without incurring runtime cost for assertions, and therefore allow the developer to follow the C motto “not paying for what you don't use” without any additional risk.

```c
#include <inttypes.h>
#include <stddef.h>
#include <stdlib.h>

typedef struct List List;
struct List { int32_t x; List* tail; };
static List* const NIL = NULL;

List* cons(int32_t x, List* tail) {
  // This might fail or take an arbitrary long time to execute.
  List* head = malloc(sizeof(List));
  head->x = x;
  head->tail = tail;
  return head;
}

void release(List* list) {
  if (list != NULL) {
    // This recursion could blow up the stack.
    release(list->tail);
    free(list);
  }
}
```

Listing 2: List of integers in C using heap allocation.

```scala
case class ArrayList(val buffer: Array[Int], var size) {
  require(0 <= size && size <= buffer.length)

  def appendSafe(x: Int): Boolean = {
    if (size < buffer.length) {
      buffer(size) = x
      size += 1
      true
    } else false
  }

  def appendOptimized(x: Int): Unit = {
    require(size < buffer.length) // Only checked statically.
    buffer(size) = x
    size += 1
  }
}
```

Listing 3: Optimisation through verification

3 Additional Features For Leon

We introduce in this section new tools for Leon that aim at increasing the safety of Scala programs, allowing Leon to handle a larger fraction of the Scala programming language and providing users with I/O operations through a safe API.

3.1 Strict Arithmetic Model & 8-bit Integer

The verification facilities of Leon were augmented in two ways. Firstly, **`Byte`** is now supported in addition to **`Int`** and **`BigInt`**. Secondly, we have introduced the `--strict-arithmetic` option to help detect unexpected or undesired behaviour in the source code in order to avoid many runtime bugs.

8-bit Integer Support

The smallest integer available in Java and Scala is **`Byte`**. It is specifically defined as a 8-bit signed integer using 2's complement representation and therefore can represent any integer in the asymmetric range $[-128, 127]$. Furthermore, arithmetic operations on **`Byte`** and **`Int`** (and any other primitive type) is fully defined by the Java specification [21].

In order to introduce into Leon support for **`Byte`** we decided to explicitly include in the Abstract Syntax Tree (AST) representing programs the different integer promotion rules defined by Java that are followed by Scala as well. This has the advantage to encode those rules in one place only (i.e. when extracting a program into its AST form) such that no other module of Leon needs detailed knowledge of the nature of integer arithmetic.

For the current scope of Leon, the integer promotion rules consist of:

1. widening **`Byte`** to **`Int`** (i.e. adding 24 leading bits such that the newly created integer represent the same value) when a **`Byte`** is involved in any arithmetic operation, and

2. narrowing **`Int`** to **`Byte`** (i.e. keeping only the 8 least-significant bits) when the integer is explicitly truncated.

The Java integer promotion rules impose that a **`Byte`** is automatically widened to an **`Int`** before carrying out any arithmetic operations, even for the bitwise negation (`~`) operator which therefore produces 32-bit integers as if one would have written `~(b.toInt)`. However, narrowing casts are always explicitly introduced with the `toByte` method of **`Int`**.

We detail in the Section 5.1 how **`Byte`** support is implemented in *GenC*.

Strict Arithmetic Checking

It is common knowledge that division by 0 in Java triggers a `java.lang.ArithmeticException` and Leon has been emitting verification issues when such a dangerous instruction could be triggered in the user code. Furthermore, Java specifies exactly how should **`Int`**`.MinValue` be transformed by the unary minus operator on the asymmetric range $[-2^{31}, 2^{31} - 1]$: the JVM should precisely yield **`Int`**`.MinValue` and not throw any exception. Because this behaviour is not obvious and could lead to bugs in a software we have introduced a more strict verification mode, available with the `--strict-arithmetic` option, that automatically instruments the input source code to detect the following potentially undesired situation (where `x` and `y` both denote **`Int`** values):

1. Overflows (also independently available with the `--overflow` option)
   (a) `-x`, when `x` is `Int.MinValue`;
   (b) `x + y`, when the values of `x` and `y` produce an overflow;
   (c) `x - y`, when the values of `x` and `y` produce an overflow;
   (d) `x * y`, when the values of `x` and `y` produce an overflow;
   (e) `x / y`, when `x` and `y` are `Int.MinValue` and `-1`, respectively;
2. Undesired behaviour
   (a) `x % y`, when `x` and `y` are `Int.MinValue` and `-1`, respectively.
   While the result is very well defined in Java and Scala and one can understandably expect an answer of 0 this is not well defined in C99 and is actually undefined in C11 [24]. This is why we treat is as undesired behaviour with the `--strict-arithmetic` option.
   (b) `x >> y`, `x >>> y` or `x << y` when `y` is outside the range$[0, 31]$.
   In Java and Scala only the 5 lowest-order bits of `y` are considered, effectively mapping `y` onto $[0, 31]$ using modulo arithmetic. Not only is the behaviour undefined in C99 when `y` is outside of this range but we believe this behaviour to be always undesired and therefore unsafe.

Although `x << y` can, and very often will, overflow we believe that this behaviour is actually intended when manipulating bit patterns. For signed integer in C99, such overflow are however undefined. We detail in Section 6.2 how we deal with this issue when generated equivalent and safe C code.

The overflows detected here can partially be detected at runtime in a Java program by using the `negateExact`, `addExact`, `substractExact`, and `multiplyExact` methods of `java.lang.Math` [25]. It however appears that no such built-in method exists for detecting overflows in divisions.

The extra assertions added by `--strict-arithmetic` are assumed to be verified when converting the Scala code into C by *GenC* in order to avoid triggering undefined behaviour when running the generated C code.

3.2 I/O Framework

While most embedded systems do not have file I/O it remains an important feature for many programs. That is why we have introduced I/O support for both standard input and output streams and files in Leon's library. We describe here the relevant features of the `leon.io` package for *GenC*.

This framework serves also as a proof of concept for using C library functions and types with *GenC*. We further discuss this point in Section 7.

`leon.io.StdOut` One case use the `StdOut` object in order to read data from the standard input stream. Specifically, it is possible to print, with or without a new line, `String`, `Char`, `Byte` and `Int` with the `print` and `println` functions.

`leon.io.StdIn` Similarly, the `StdIn` object provides facilities to read data from the standard input stream. For reasons exposed in Section 5.1, it is currently only possible to read `Int` and `Byte`.

Two flavours are available: `readInt` (`readByte`) and `tryReadInt` (`tryReadByte`). The first one always returns a number and therefore should only be used when one can trust the input source to be valid. The other one returns a `leon.lang.Option` in order to distinguish success reads from failures such as reaching the end of the input or, in the case of `tryReadInt`, attempting to read something that doesn't match the format of an **`Int`**.

In order to provide equivalence between C and Scala we implement those functions such that they follow the C conventions on byte and 32-bit decimal integer format, skipping any leading space for the latter as `scanf("%d", &x)` would assuming 32-bit integers. That is, we accept integers that match the following regular expression: `\s*[+-]?\d+`. For simplicity reasons, we leave handling of overflows as undefined.

A curious reader might wonder why the provided functions take an implicit `State` parameter: `def readByte()(implicit state: State): Byte = ...`. This is an artefact for verification purposes [8]: it ensures that calling such methods can produce different results every time.

`leon.io.FileOutputStream` To print content to a new file, we provide `FileOutputStream`. It provides functions to check whether the file was successfully opened and the ability to write **`String`**, **`Char`**, **`Byte`** and **`Int`** at the end of the file through the `write` overloaded method which requires the stream to be open and report errors by returning **`false`**.

It is important to note that one should always invoke the `close()` method when the file stream is no longer needed in order to let the underlying system reclaim file resources.

`leon.io.FileInputStream` Like `StdIn`, `FileInputStream` provides an API to read **`Byte`** and **`Int`**, but from a file. The `readByte`, `readInt`, `tryReadByte` and `tryReadInt` methods follows the same convention as detailed above, plus require that the file is open. Finally, one should also call `close()` when the file stream is no longer used to avoid leaking resources.

4 Memory, Aliasing & Mutability Models For *GenC*

In the previous section we discussed how Leon was extended to support more Scala programs with additional verification conditions to enforce a higher quality of software. We now continue the discussion of overall feature support in Leon/*GenC* by exposing the memory model used by the generated C code, and which limitations we impose in addition of the restrictions required by Leon.

Thanks to the *xlang* module of Leon it is possible to transform a large set of imperative programs into a functional equivalent and to verify such programs. With *GenC* we however want to keep the imperative nature of the input code in order to produce as efficient as possible C code. To achieve this we do not apply this transformation to the Abstract Syntax Tree (AST), yet we retain the same restrictions on the supported imperative features because one of the requirement of *GenC* is that the input program must be valid and therefore verified.

On the one hand, *xlang* allows one to use local `var`ibables, assignments, blocks, `while` loops (with `invariant`), imperative arrays (i.e. array elements can be mutated) and mutable objects (i.e. classes with fields declared using `var`).

On the other hand, it forbids most form of aliasing: in any given scope no two named objects (i.e. local variable, function parameter or class member) can refer to the same memory area. To ensure no aliasing occurs, it is also not possible to return from a function one of its mutable parameters. Listing 4 gives an example of illegal program where a variable is aliased.

```scala
case class Counter(var i: Int)

def scope {
  val c = Counter(0)

  def foobar(d: Counter) {
    d.i += 1
    c.i += 1
  }

  foobar(c) // ERROR: c is aliased
}
```

Listing 4: Example of an illegal program exhibiting aliasing.

Such restrictions do not apply on immutable objects because it does not change the behaviour of the program to copy immutable class instances instead of sharing references to a single object.

Consequently, mutable objects have *a single static allocation point* and, when passed to a function, a reference can be used to identify the object. If the mutable object is local to a function it can be returned; at which point the object can be copied into a new instance at the call site, without breaking existing references as there are none, as we illustrate in Listing 5.

Additional, Leon disallows the usage of `null`. Hence, when converting code with mutable objects in C99 we can safely represent references using pointers without having to deal with `NULL` pointer dereferencing. It also follows from the no-aliasing rules that at no point a memory location can be accessed after it was released and therefore let us avoid a whole category of potential bugs. This means that, in order to respect the equivalence between the produced C code and the input Scala code, we do not need to implement any kind of garbage collector

```
case class Counter(var i: Int)

def fun: Counter = {
  val c = Counter(0)
  // ...
  c.i += 1
  // ...
  c
}

def gun {
  val c = fun // Copy the returned Counter.
  c.i += 1
}
```

Listing 5: Returning mutable object is safe for local variables.

because all objects use the *automatic* storage class[1]: they live on the stack and are destroyed when the program exits their scope, releasing their memory in a deterministic order. Thus no heap memory is needed throughout the execution of the transpiled program.

There is however a limitation to this general framework: because of the nature of arrays allocated on the stack in C it is not possible to return them from functions. We detail the underlying reasons for this limitation in Section 5.2.

As mentioned above, immutable objects can be copied without impacting the behaviour of the program. It therefore remains to be defined when an object is considered mutable in a given context.

A class is considered to be mutable if any of its fields is declared with the `var` keyword or if any fields has a mutable type; additionally, any class of a hierarchy is considered as mutable type if there exists at least one class in the hierarchy that is defined as mutable under the previous rules. A specific `Tuple` type is only considered mutable if any of its member is of mutable type, but arrays are never considered mutable because they already propagate side-effects even when duplicated thanks to the indirection used in their representation. Primitive types, such as `Int` of `Byte`, and `String` are not considered mutable.

Finally, a nested function parameter list is extended to include variables available in its scope as if they were mutable objects in order to handle side-effects properly.

We illustrate how these rules are applied in Listings 6 and 7; the specifics of basic types, classes and functions translation are detailed in Section 5.1, Section 5.3 and Section 5.4, respectively.

However, this model suffers from one limitation we exemplify in Listing 8 that is considered valid by *xlang* but not by *GenC*.

The issue is that, with static allocations, when an instance of `B` is constructed it needs to own all its members or the instance cannot be returned at all from any function: if we were to translate `B` to `struct B { A* a; };` instead of `struct B { A a; };` then the construction of a `B` would need two static allocation points, one for the member `a` and one for the constructed `B`, and upon returning from a function the member `a` would be pointing to an invalid location and accessing it would result in undefined-behaviour, as illustrated by Listing 9.

[1]See Section 6.2.4 *Storage durations of objects* of the C99 Standard for a full definition of the different storage classes.

```
case class A(var x: Int) // mutable type

def scope: A = {
  var i = 0

  def updateAndInc(a: A) {
    i += 1   // might overflow
    a.x += i // might overflow
  }

  val a = A(41)
  updateAndInc(a)
  assert(a.x == 42)

  a
}
```

Listing 6: Example of code with mutable type.

```
typedef struct { int32_t x_0; } A_0;

static void updateAndInc_1(int32_t* i_18, A_0* a_4) {
    *i_18 = (*i_18) + 1;
    a_4->x_0 = a_4->x_0 + (*i_18);
}

static A_0 scope_1(void) {
    int32_t i_18 = 0;
    A_0 a_5 = (A_0) { .x_0 = 41 };
    updateAndInc_1(&i_18, &a_5);
    return a_5;
}
```

Listing 7: C99 code generated for Listing 6.

As we expect returning mutable objects to be significantly more frequent than constructing anonymous and temporary mutable objects passed to functions, *GenC* considers that mutable types own all their members. As a consequence, the `update` function shown above, despite taking `a` by reference, cannot properly mutate `a` as a copy is sent to `updateB`. Listing 10 depicts this issue. We sketch in Section 10 how this limitation might be lifted.

```scala
case class A(var x: Int) // mutable type
case class B(a: A)       // mutable type

def updateB(b: B): Unit = {
  b.a.x = 42
}

def update(a: A): Unit = {
  updateB(B(a)) // Note: the constructed B is an anonymous object
} ensuring(_ => a.x == 42)
```

Listing 8: Example of code accepted by Leon but illegal with *GenC*.

```c
typedef struct { int x; } A;
typedef struct { A* a; } B;

/* Dangerous translation of: def foobar = B(A(42)) */
B foobar(void) {
  A tmp = (A) { .x = 42 };
  B b = (B) { .a = &tmp };
  return b;
} /* b.a is now a dangling pointer */
```

Listing 9: Example of a dangling pointer bug in C.

```c
typedef struct { int x; } A;
typedef struct { A a; } B;

static void update(A* a) {
  updateB(&(B) { .a = (*a) });
  /* copy a:           ^^^^  */
}
```

Listing 10: Example of the limitation of *GenC* memory model.

5 Translation of Definitions

With the memory model and its limitations defined in the previous section we now start discussing how definitions are translated from Scala to C.

Support for primitive types, arrays, classes as well as functions has been improved since the initial release of *GenC*. We describe here what is supported and how the translation from Scala to C is performed at a high level, and, of course, what new features *GenC* has in store.

5.1 Primitive Types & Literals

Support for `Unit`, `Boolean` and `Int` is unchanged since their introduction in the previous version of *GenC*; that is, they are directly mapped to `void`, `bool` and `int32_t`, respectively. In addition to those types we add the possibility to use `Byte` in the input programs, mapping this type to `int8_t`. The `Unit` literal `()` disappears during the C translation, but `true`, `false` and the integer literals are mapped into equivalent, decimal literals.

The conversion from `Byte` to `Int`, or vice versa, is handled through explicit cast in C. Casting from 8-bit signed integer to 32-bit signed integer is always safe in C because any value representable with 8 bits is representable with more than 8 bits. In contrast to that, converting an integer that lies outside the $[-128, 127]$ range falls into implementation-defined land. As discussed in the Section 8 we require that the user C compiler must use 2's complement representation for integers and when casting a value from one integral type to another a modulo arithmetic is used. In the particular conversion from `int32_t` to `int8_t` we expect C compilers to, in effect, keep only the 8 lowest-order bits.

In addition to the above primitive types we introduce a minimal support for `Char` and `String`. Again, the Java standard defines in great details how characters and sequence of characters handle their different operations, and especially what encoding is supported. But the C99 standard is very lax on this topic and leaves many aspects up to the implementer, which makes it extremely difficult to write truly portable C code involving string manipulations. That is why *GenC* currently limits the support of characters and strings to raw ASCII literals.

One advantage of such a restrictive model is that, because in C99 string literal have a static storage, it is legal for functions to return them. Moreover, converting the `String` type to `char*` is valid thanks to the fact strings cannot be mutated. Obviously, this comes at the cost of not being able to read strings from files or through `StdIn`. `Char` cannot yet be read from external sources as well because of the complexity of encoding. Furthermore, because character literals are limited to the ASCII range range, we are allowed to use `char` as a valid representation of such values on any platform.

5.2 Tuples & Arrays

Support for `TupleN[T1, ..., TN]`, or `(T1, ..., TN)`, is virtually unchanged since the initial release of *GenC*: a specific C structure gets generated for any combination of type parameters as they appears in the input source code with fields names `_1` to `_N`.

Similarly, support for `Array[T]` was not significantly modified. It is modelled in two parts: an allocation point that reserves the memory on the stack using a classic C array, or a *Variable Length Array* (VLA) when the array is initialised with `Array.fill(length)(value)` where `length` is not an integer literal, of the corresponding type and, secondly, a C structure that keeps track of the beginning of the memory reserved for the array using a pointer and the length of the array, because this information is not present in the Scala `Array` type although the array cannot be resized at runtime.

While most compilers do support VLA, MISRA guidelines requires that they shall not be used. We leave this responsibility to the user but do emit a warning when such an array is generated by *GenC*.

With respect to the memory model, each array is statically allocated on the stack, VLA or not. This has for consequence that returning the structure representing an array from functions is not possible as this would result in dangling pointers. *GenC* also prevents you from returning tuples or objects that themselves contain arrays for the same reason. Unfortunately, the C language doesn't provide tools to workaround this limitation without using heap allocation. It is however possible to inline a function within Leon, using the `@inline` annotation, but one should refrain from abusing of this technique.

```
typedef struct { int32_t _1; bool _2; } Tuple_int32_bool;
typedef struct { Tuple_int32_bool* data; int32_t length; }
        array_Tuple_int32_bool;
```

Listing 11: C data structure representing `Array[(Int, Boolean)]`.

5.3 Classes

Support for classes was significantly extended with this new version of *GenC*. Previously, only case class having immutable fields and no inheritance nor generics were supported. Those limitations are now lifted but types shall still be non-recursive. For example,

```
abstract class Expr
case class Add(lhs: Expr, rhs: Expr) extends Expr
```

will be rejected by *GenC* because `Expr` and `Add` require each other to be fully defined and this is not possible in C without the indirection of pointers and multiple allocation points. Two additional restrictions over Scala that Leon imposes are that fields can only be defined in case classes and that abstract classes must have at least one concrete child.

When no inheritance is involved, case classes and their fields are mapped to a C structure with matching types. In order to support mutability, fields are not marked as **`const`**. When inheritance is involved, the full class hierarchy is considered together: each case class is mapped to a dedicated C **`struct`** as if no inheritance was involved, and the top level, abstract class is translated to a *tagged union* where the tag is a C **`enum`** that identifies the runtime type within the class hierarchy while the actual object is stored in a C **`union`** of all possible case classes. In all cases, **`union`** and **`struct`** are initialised using the *designated initialiser lists* syntax.

As one constructs an instance of a class the runtime type is always known. But this information can get lost when passing the object to a function. And so any function could apply pattern matching or test for membership on the passed object. In order to preserve this information we represent any object of a class hierarchy with a tagged union.

MISRA advises against using **`union`**, arguing that it is hard to use correctly as it can involve implementation-defined behaviour when a union member is read after a different member was written. However, in this particular case, thanks to Scala typing system, no two different union members will get read or written during the lifetime of an object because the type of an object cannot be changed and because all cast are safe if the program verification went successfully.

With this model, one can easily check using `isInstanceOf` whether an object of a given class hierarchy is, at runtime, any of the case classes. However, one cannot directly known if an object is an instance of a non top level abstract class; instead one has to check if the object's tag denotes any of the concrete case classed derived from the given abstract class. This process is not yet automated by *GenC*. The same limitation applies to casts: it is not possible to cast an object to an intermediate level of the hierarchy. This, on the other hand, as no negative consequence because an object is always represented by the same tagged-union.

An upside of this model is that it does not suffer from the slicing effect that many C++ developers have faced when returning an object by value. Or course, this is possible only because we use as much memory as the widest case class in the hierarchy for all instances, regardless of their runtime type.

Contrary to Scala, C99 disallows **`struct`** and **`union`** to have no members at all. For the latter, because Leon requires that all class hierarchies have at least one concrete member, this is not an issue. In order to produce valid code, *GenC* will introduce a dummy byte member in every empty class.

But unlike Scala, C provides a language feature to define enumeration. The idiomatic way to represent a small domain of values in Scala is using inheritance and empty case classes. For that reason *GenC* maps to an **`enum`** any class hierarchy for which all case classes have no field.

Finally, generics are now supported as well. A notable difference between Scala Generics, as they are handled by the JVM, and Generics within Leon is that Leon does not do type erasure for verification purposes. This difference is very handy for *GenC* as it allows us to see meta-types as C++ template classes: for each combination of type parameters used in the program, a specific type is generated at build time.

Listing 12 shows an example of class inheritance with polymorphic function and Listing 13 reports the code generated by *GenC* for this particular example.

5.4 Functions

In addition to the support previously introduced for regular, overloaded and nested functions with call-by-value arguments, *GenC* now handles generic functions in a similar fashion to generic types: they are not typed erased in Leon which allows us to consider them as templates for specialised functions.

For safety reasons, MISRA imposes that functions shall not call themselves, either directly or indirectly. For that reason *GenC* will emit a warning when a recursive function is used. The guidelines, however, allow such functions to be used if one can determine the worst case stack space required during execution. And, as a matter of fact, the ecosystem of Leon offers support for resource bound inference [26].

With a few exceptions listed in Section 7 higher-order functions are not supported and functions are not regarded as first-class citizens.

GenC expects two specific functions to be present in the input source: `main` and `_main` inside the main object that represents the program. Those functions must follow a specific format. The first one needs to match Scala convention for obvious compatibility reasons, but also must be declared as `@extern` in order to let *GenC* ignore the mandatory array of strings which are currently not well supported. In Java and Scala, the `main` function returns nothing but *GenC* nonetheless converts it to a C function that returns an integral value assumed to be in the range of an octet[2]. This value is obtained by calling the second function, `_main`, with no parameter.

[2]This is because most operating systems ignore all but the 8 least-significant bits.

```scala
abstract class Top

case class One(x: Int) extends Top

abstract class Derived extends Top {
  def sum: Int
}

case class Two(x: Int, y: Int) extends Derived {
  def sum: Int = x + y // can overflow
}

case class Three(x: Int, y: Int, z: Int) extends Derived {
  def sum: Int = x + y + z // can overflow
}

def foo = {
  val x = One(1)
  val y = Two(1, 2)
  bar(x, y)
}

def bar(one: One, d: Derived) = one.x + d.sum // can overflow
```

Listing 12: Example of class inheritance and polymorphic function in Scala

To sum up, the core of the program must be similar to the following:

```scala
import leon.annotation.extern

object Program {
  @extern
  def main(args: Array[String]): Unit = _main()

  def _main(): Int = {
    leon.io.StdOut.println("Hello, World!")
    0
  }
}
```

When converting such programs, *GenC* identifies all the dependencies of `_main` (i.e. the set of functions and types involved) without considering loop invariants, assertions, pre- and postconditions and transpiles only these dependencies.

```c
typedef enum { tag_Two_0, tag_Three_0, tag_One_0 } enum_Top_0;
typedef struct { int32_t x_1; int32_t y_0; } Two_0;
typedef struct { int32_t x_2; int32_t y_1; int32_t z_0; } Three_0;
typedef struct { int32_t x_0; } One_0;

typedef union { Two_0 Two_0_v; Three_0 Three_0_v; One_0 One_0_v; }
        union_Top_0;

typedef struct { enum_Top_0 tag; union_Top_0 value; } Top_0;

static int32_t bar_1(Top_0 one_0, Top_0 d_0) {
    int32_t norm_1 = one_0.value.One_0_v.x_0;
    int32_t norm_0 = sum_2(d_0);
    int32_t norm_2 = norm_0;
    return norm_1 + norm_2;
}

static int32_t sum_2(Top_0 thiss_0) {
    if (thiss_0.tag == tag_Three_0) {
        return (thiss_0.value.Three_0_v.x_2 + thiss_0.value.Three_0_v.y_1) +
                  thiss_0.value.Three_0_v.z_0;
    } else if (thiss_0.tag == tag_Two_0) {
        return thiss_0.value.Two_0_v.x_1 + thiss_0.value.Two_0_v.y_0;
    }
}

static int32_t foo_1(void) {
    Top_0 x_27 = (Top_0) {
      .tag = tag_One_0,
      .value = (union_Top_0) { .One_0_v = (One_0) { .x_0 = 1 } }
    };
    Top_0 y_8 = (Top_0) {
      .tag = tag_Two_0,
      .value = (union_Top_0) { .Two_0_v = (Two_0) { .x_1 = 1, .y_0 = 2 } }
    };
    return bar_1(x_27, y_8);
}
```

Listing 13: Generate C code for Listing 12.

6 Translation of Expressions

We exposed in the previous section which definitions are supported and how they are translated into C99. We continue in this section to describe the supported fragment of Scala by focusing on expressions.

This new version of *GenC* improves some arithmetic operations and uses a more strict normalisation of instructions, but also adds support for pattern matching. Conditional and loops statements, **`if`** and **`while`**, were already introduced in the previous iteration of the project, and assertions were not translated into C code.

Additionally, *GenC* now properly rejects code using the `==` relational operator for case classes in order to focus our efforts on more critical constructs that cannot be implemented manually in the user codebase using a regular function.

6.1 Execution Order Normalisation

On the one hand, Scala defines a strict evaluation order for expressions which is based on the model used by Java: expressions are evaluated from left to right, with a few exceptions such as short circuiting in || and `&&` operations. On the other hand, C provides a lax execution policy: between two *sequence points*[3] expressions can be evaluated in an arbitrary order. Hence, the same code can produce different results depending on the platform or compiler.

In order to produce programs that have exactly the same behaviour than their source, we make sure that enough *sequence points* are explicitly introduced by means of temporary variables. With this new release we are enforcing a stricter behaviour to produce valid code even for snippet like Listing 14, which is transpiled into Listing 15.

```
// Extract from regression test: ExpressionOrder.scala
def test9() = {
  var c = 0
  val r = { c = c + 3; c } + { c = c + 1; c } * { c = c * 2; c }
  r == 35 && c == 8
}.holds
```

Listing 14: Example of complex Scala code supported by *GenC*.

Regarding performance, using any modern compiler with its basic set of optimisations (`-O1`) should be enough to make most if not all intermediate results disappear from the generated assembly instructions, without involving what are considered for some dangerous optimisations.

6.2 Bitwise Shift Operators

As reported in the Appendix A, which details the behaviour of all arithmetic operators under the C and Java standards, most of the operators supported previously by *GenC* were correctly implemented. However, this was no the case for the bitwise shift operators, mainly because signed shift operations do not work as one might expect in C, we have to rely on the unsigned version of `<<` and `>>` to match the behaviour of Java's `<<` and `>>>` operators.

However, this means converting **`int32_t`** to **`uint32_t`** and back. The first conversion is well defined in C99: a positive integer is represented correctly as an unsigned integer, and a

[3]See Section 5.1.2.3 *Program execution* of the C99 Standard for the full definition of *sequence points*.

```
static bool test9_1(void) {
    int32_t c_13 = 0;
    c_13 = c_13 + 3;
    int32_t norm_121 = c_13;
    c_13 = c_13 + 1;
    int32_t norm_119 = c_13;
    c_13 = c_13 * 2;
    int32_t norm_120 = c_13;
    int32_t norm_122 = norm_119 * norm_120;
    int32_t r_3 = norm_121 + norm_122;
    return r_3 == 35 && c_13 == 8;
}
```

Listing 15: Code generated for Listing 14

negative integer gets increased by 2^{32} for the conversion. The opposite conversion is trickier: if the unsigned integer falls in the range of values represented by the `int32_t` type then the conversion is safe; otherwise, it is implementation-defined. Because of this, we have two options:

1. Expect the C compiler to properly define the `uint32_t` to `int32_t` as a "simple" *reinterpret* cast of the underlying binary representation as a 2's complement representation.
2. Use unsigned integer everywhere else to do only initial `int32_t` to `uint32_t` conversions. This, of course, implies handling every operations with unsigned integers. For arithmetic operations, this seems fine. For relational/logical operators such as `<`, this means testing for sign bit manually in addition of the same operator on unsigned integer.

At this point, the first solution seems better for two reasons. Firstly, it impacts only two operators with no measurable performance impact, assuming casts are in the worst case scenario moving data from one specific register kind to a register of another kind, while additional bit checking would have an impact on performance (despite being small and probably irrelevant for many applications, this could be a concern for embedded devices). Secondly, if one cannot ensure that the C compiler indeed fulfils our assumptions, one could devise a safe conversion function, at a runtime cost.

After inspection of compilers' manual it appears that GCC [27] and Microsoft's [28] compilers match the assumed behaviour; clang lacks such information but we can assume GCC's model to hold as well [29].

6.3 Integer Conversions

With the support for `Byte` we also introduce the ability to convert an `Int` to a `Byte` by means of Scala's explicit casts with the `toByte` method, as well as the reverse operation though implicit casts, which arise from Java's integer promotion rules, or explicit casts when invoking `toInt` on a `Byte`.

The C99 standard specifies that, when converting from one integral type to another, if the given value can be represented by the destination type then the conversion is safe, however if the value is not in the destination range then the conversion is implementation-defined.

Therefore, the widening conversion from `Byte` (`int8_t`) to `Int` (`int32_t`) is not an issue. On the other hand, the narrowing cast from `Int` to `Byte` will fall outside the portable fragment of C for many values. Similarly to *GenC*'s conversion of `>>>` we decided to rely on C implementations to obey the most common truncating behaviour but, were this requirement not met, one could also update the translation to impose the Java arithmetic model in C by using additional operations at a performance cost.

6.4 Membership Test & Cast

We leverage the *tagged-union* used to represent inheritance in order to support membership test. Indeed, we only need to compare the tag of an instance to the tag of the given type. Referring to the classes defined in Listing 12 from Section 5.3, the following Scala code

```
def fun(b: Top) = b.isInstanceOf[One]
```

gets translated into:

```
static bool fun(Top b) { return b.tag == tag_One; }
```

Similarly, we fully take advantage of the memory storage to perform casts, for which Leon provides tools to prevent all illegal casts, as we show in the next listing. Concretely, *GenC* transpiles casts into accesses to the appropriate **`union`** member. For example, *GenC* transpiles

```
def gun(b: Top) = {
  require(b.isInstanceOf[One]) // Ensures that the cast is legal
  b.asInstanceOf[One].x
}
```

into the following equivalent C99 code:

```
static int32_t gun(Top b) { return b.value.One_v.x; }
```

With our model for inheritance it is however not possible to directly test membership for intermediary abstract class, such as `Derived` in Listing 12, or perform a cast to an abstract class, with the exception of the root class in a hierarchy as this can be regarded as a NOP. *GenC* therefore forbids such instructions.

The `isInstanceOf` and `asInstanceOf` expressions are used internally to handle some forms of pattern matching, which we present next.

6.5 Pattern Matching

Support for most forms of pattern matching is also introduced in this new version. That is, with the exception of the *Unapply Pattern*, also known as *Extractor Objects*, pattern matching expressions are converted to imperative expressions using **`if-else`** statements.

The format of a pattern matching expressions is

```
scrutinee match {
  case pattern1 [if guard1] => body1
  // ...
  case patternN [if guardN] => bodyN
}
```

where `scrutinee`, `body1`, . . . , `bodyN` are expressions, each pattern guard is an optional expression that has no side-effect and `pattern1` to `patternN` can be any of following kind of patterns:

Binder Pattern binds the `scrutinee` to a given name `b` when `b` is not the wildcard _. The identifier `b` can then be used in the pattern guard and body expressions, essentially aliasing `scrutinee`.

Literal Pattern with an optional binder, checks whether the `scrutinee` is equal to a given literal `l`, which type must match at compile time the type of `scrutinee`.

Instance Of Pattern checks whether the `scrutinee` is of a given type `T`, with an optional binder as described above. When the binder is used in the pattern body or guard it has the same type `T` as if `scrutinee` was casted to `T`. Note that `T` has to be related to the static type of `scrutinee` in order to make the program type check.

Tuple Pattern in addition to an optional binder it recursively checks the given sub-patterns for each member of the tuple.

Case Class Pattern binds the `scrutinee` to the given optional binder, checks that it is of the given class type and recursively applies the given patterns for each class field.

Because the guards and patterns are assumed to have no side-effect we need only to ensure the `scrutinee` is a constant expression. In order to do that, we can create a temporary variable if it is, for example, a function call. That has the benefit of allowing us to replace the binders from both guards and bodies with an expression of our design that can prevent variable to be copied and therefore to incorrectly propagate side-effects when doing the C translation of the resulting if-else expressions, as we illustrate with Listings 16, 17 and 18.

In addition to the classic **`match`** form of pattern matching, Leon and *GenC* also support the *let* pattern matching, such as `val (a, b) = getPair()` or `val Some(x) = getSome()`, using the same conversion technique.

```scala
case class A(var x: Int)

abstract class Base
case class Derived1(a: A) extends Base
case class Derived2(a: A) extends Base

def foo(b: Base): Unit = b match {
  case Derived1(a)              => a.x = 100
  case Derived2(a) if a.x == 42 => a.x = 58
  case Derived2(a)              => a.x = 0
}
```

Listing 16: Example of regular pattern matching in Scala.

```scala
case class A(var x: Int)

abstract class Base
case class Derived1(a: A) extends Base
case class Derived2(a: A) extends Base

def foo(b: Base): Unit = {
  if (b.isInstanceOf[Derived1]) { b.asInstanceOf[Derived1].a.x = 100 }
  else if (b.isInstanceOf[Derived2] && b.asInstanceOf[Derived2].a.x == 42) {
    b.asInstanceOf[Derived2].a.x = 58
  } else if (b.isInstanceOf[Derived2]) { b.asInstanceOf[Derived].a.x = 0 }
}
```

Listing 17: Conversion of pattern matching from Listing 16 into equivalent **if-else** expressions.

```c
typedef struct { int32_t x_0; } A_0;

typedef struct { A_0 a_0; } Derived1_0;
typedef struct { A_0 a_1; } Derived2_0;

typedef enum { tag_Derived1_0, tag_Derived2_0 } enum_Base_0;
typedef union { Derived1_0 Derived1_0_v; Derived2_0 Derived2_0_v; }
        union_Base_0;
typedef struct { enum_Base_0 tag; union_Base_0 value; } Base_0;

static void foo_1(Base_0* b_0) {
    if (b_0->tag == tag_Derived1_0) {
        b_0->value.Derived1_0_v.a_0.x_0 = 100;
    } else if (b_0->tag == tag_Derived2_0 &&
               b_0->value.Derived2_0_v.a_1.x_0 == 42) {
        b_0->value.Derived2_0_v.a_1.x_0 = 58;
    } else if (b_0->tag == tag_Derived2_0) {
        b_0->value.Derived2_0_v.a_1.x_0 = 0;
    }
}
```

Listing 18: Generated C99 code for Listing 16 based on the transformation exposed in Listing 17.

7 Library Support Using Annotations

Now that we have detailed how language constructs can be automatically converted into C we discuss how one can workaround some limitations and also how external C libraries, such as the Standard C Library, can be integrated to work with verified Scala code.

Although *GenC* does not support higher-order functions at the moment, Leon's library was updated so that most of `leon.lang.Option` methods can be used by program translated into C. The trick used here is simply to annotate those methods with `@inline` in order to remove the presence of lambdas in the AST. One can then use `getOrElse`, `orElse`, `map`, `flatMap`, `filter`, `withFilter`, `forall` and `exists`. This trick can be used in the user code as well but is limited, for obvious reasons, to non-recursive higher-order functions.

When it comes to functions using system calls, such as I/Os, no automated conversion is possible. In those situations we let the user define his own implementation for functions, add manual conversions from Scala types to C types or even drop some functions and types from the translation, with `@cCode.function`, `@cCode.typedef` and `@cCode.drop` annotations, respectively, from the package `leon.annotation`. Their usage is detailed in the following subsections.

Generally speaking, those annotations can be used to make proxies between the verified, Scala code and some existing, trusted C libraries.

7.1 Custom Function Implementation

In order to circumvent some current limitations of Leon/*GenC* ecosystem one can use the

```
@cCode.function(code, includes)
```

annotation, usually accompanied by `@extern`, to define the corresponding implementation of any function or method. The parameters are meant to be used as follows:

`code` For convenience, the C implementation given by `code` is represented using a string literal and not an Abstract Syntax Tree. The user is responsible for the correctness of the provided C99 code.

Because *GenC* renames the functions, e.g. to deal with overloading, the special `__FUNCTION__` token should be used instead of the original name. Furthermore, the parameters and return type should match the signature automatically generated by *GenC*.

For top-level functions the argument list is the same in C, modulo the translated types. For nested functions, one has to add extra parameters for the variable available in the function scope. And finally, for methods an extra argument for **`this`** needs to be added as well. In case of mistakes in the parameters the user can refer to the automatically generated function declaration, update the annotation and re-run *GenC*.

`includes` The optional parameter `includes` can hold a colon separated list of the required C99 include header files for the implementation of the function. *GenC* makes sure to include headers exactly once, for readability purposes.

Here is a typical example of a logging function using `<stdio.h>` facilities:

```
// Print a 32-bit integer using the *correct*
// format for printf in C99
@cCode.function(
  code = """
    |void __FUNCTION__(int32_t x) {
    |  printf("%"PRIi32, x);
    |}
    """,
  includes = "inttypes.h:stdio.h"
)
def log(x: Int): Unit = {
  print(x)
}
```

7.2 Custom Type Translation

```
@cCode.typedef(alias, include)
```

can be used when a whole type needs to be represented using a special C type. Here the `include` parameter is also optional, however it can only refer to one header as it is not expected to have a type defined in several headers. The `alias` string must represent an existing and valid C type. As a consequence, the annotated type members do not get translated into equivalent C code automatically.

Using an aliasing from `T` to `U` allows the user to use `U` directly when manually defining the implementation of functions instead of having to deal with unique identifiers issues.

This annotation can be used, for example, to build a specific translation of a file API:

```
@cCode.typedef(alias = "FILE*", include = "stdio.h")
case class MyFile(// ...
```

7.3 Ignoring Functions or Types

It is also possible to skip the translation of some functions or types that are only used as implementation details in proofs using the

```
@cCode.drop()
```

annotation. As *GenC* triggers an error if a dropped type or function it can be useful to ensure that the type or function is indeed not used in live code.

8 Toward MISRA Compliance

It is well-known that C is a popular, powerful but complex language, and as any programming language it has its imperfections. With that in mind, the *Motor Industry Software Reliability Association* (MISRA) [22] strives to provide and promote best practice in developing safety-related embedded electronic systems and other software-intensive applications by means of guidelines and requirements. The 2012 edition of the guidelines helps define a subset of the C90 and C99 languages that not only avoids features relying on undefined, unspecified or implementation-defined behaviour to ease portability and maintenance of programs, but also attempts to avoid obscure constructs easily misused or misunderstood and make sure that runtime errors are handled appropriately. To do that, MISRA lays down the ground rules for a software development process that impacts the selection of tools – such as which C language standard, compilers and analysis tools – the definitions of the software requirements, especially the ones about safety, and design specifications. In addition, it specifies how a team can claim compliance for a given project through process activities required or expected by MISRA, or, if needed, how a deviation to the guidelines should be documented.

GenC generates code that was engineered to follow most of the directives and rules imposed by MISRA guidelines for C99 with its Appendix E *Applicability to automatically generated code* in mind. But, for a few of them, it actually deviates from the recommendations for what we believe to be valid reasons. We discuss here how *GenC* works toward fulfilling most of the guidelines and clearly expose which ones are not met directly and for which the user should pay special attention while developing software.

The directives and rules of the guidelines are each labelled with a severity level, ranging from *mandatory*, *required*, *advisory*, to *readability*. From the many rules in the guidelines several are not relevant for *GenC* because we do not use language features concerned by those rules, such as macros or pointer type conversions. *GenC* abides by 64 rules: we focus in Section 8.2 on the 10 rules that are marked as *mandatory*, next we discuss in Section 8.3 how verification subsumes 6 rules, and in Section 8.4 we cover many rules that are met by carefully generating C99 compliant code. In Section 8.5 we discuss 5 rules that are either ignored or not met, and in Section 8.6 we highlight 20 rules that are the responsibility of the user to respect. But first we detail the general requirement for the generated C code.

8.1 General Directives & Defining a Standard C Environment

MISRA mandates that for each project a specific variant of the C language as well as requirements for the compiler implementation(s) should be defined. With *GenC* we ask the user to use a C99 compiler to compile the generated code, but add no requirement on language extensions as none are used.

The MISRA Directive 1.1 asks that all implementation-defined behaviour on which the program relies upon must be documented and understood. As exposed in the relevant sections of this report, *GenC* relies on:

1. the availability of `int32_t` and `uint32_t` types, which are defined as precisely 32-bit integers and, for the former, uses two's complement representation, and
2. converting a `uint32_t` to `int32_t` corresponds to the reinterpretation of their respective binary representations, and, finally,
3. a *byte* having 8 bits.

GenC does not rely on any other implementation-defined behaviour listed under the annex G of the MISRA Guidelines. Moreover, no assembly instructions are embedded in the produced C code, making it as platform-independent as possible.

Furthermore, provided that the input Scala code was thoroughly verified with `--strict-arithmetic` the C code should not result in undefined-behaviour, crash or exit the program except by returning from the `main` function, unless stack space was exceeded. Specifically, runtime errors mentioned in Directive 4.1 are avoided for the following reasons: 1) arithmetic errors are avoided thanks to Leon's verification capabilities, 2) pointer arithmetic is not used, 3) array bounds errors cannot occur if, again, the program was verified, 4) function parameters are always valid when Leon asserts that all function calls are valid with respect to function requirements, 5) pointer dereferencing is assumed to be valid under out memory model even tough we do not explicitly check for `NULL` pointers as explained in Section 4, and 6) dynamic memory is avoided altogether thanks to our memory model.

8.2 Ensuring Mandatory Rules

The MISRA Guidelines defines 10 rules of level *mandatory*. Three of those are not relevant for *GenC*: the first has to do with the **`sizeof`** operator (Rule 13.6), which we do not use, the second is about dynamic memory block deallocation (Rule 22.2) and the third has to do with an uncommon usage of the **`static`** keyword and array size that we do not use at all (Rule 17.6). The other 7 rules are about:

1. ensuring variables are not read before being initialised (Rule 9.1);
2. not implicitly declaring functions (Rule 17.3);
3. having an explicit **`return`** statement for all exit paths in functions (Rule 17.4);
4. not assigning or copying values to overlapping object (e.g. **`union`**) (Rule 19.1);
5. never writing to a stream that was opened as read-only (Rule 22.4);
6. not dereferencing a `FILE` object (Rule 22.5);
7. preventing using a stream after it was closed (Rule 22.6).

Points 1 is ensured by Scala automatically. Points 2 is guaranteed simply by generating explicit declaration for all functions. Point 3 is fulfilled by Scala and by ensuring all pattern matching cases are handled in a **`match`** expression. Point 4 holds thanks to the exception listed in Rule 19.1 allowing *assignment between two objects that overlap exactly and have compatible types*. Point 5, 6 & 7 are all ensured by the design of the new I/O component of Leon's library.

8.3 Rules Subsumed Under Verification

Thanks to the static analysis capabilities of Leon it is possible to verify whether some properties are respected or not, providing a counter example that explains when and why a property does not hold. For part of MISRA Guidelines we are able to take advantage of this system to notify the user when a rule or directive is not respected.

We discuss here which rules can be asserted and why:

1. As mentioned before, runtime failure are minimised to meet Directive 4.1 by identifying when an arithmetic expression could trigger undesired effect such as overflow or by ensuring all indexes used for array accesses are within valid ranges, therefore avoiding an underestimated number of bugs as discussed in Section 2.1.

```
def dot(a: Array[Int], b: Array[Int]): Int = {
  var sum = 0
  var i = 0
  while (i < a.length) {
    sum += a(i) * b(i) // overflow + invalid access
    i += 1
  }
  sum
}
```

Listing 19: Buggy implementation of the dot product.

   For example, the code in Listing 19 contains two issues: arrays `a` and `b` are not required to have the same length and the multiplication and addition operations can overflow. One can take several approaches to solve these problems. One of these consists in adding runtime checks to detect and report such problems before they happen. This of course results in performance penalty but can be sufficient for some applications. Alternatively, one can take advantage of the verification system to ensure that both arrays have always the same size and that all integers are in a specific range that prohibits integer overflow in this context, moving the burden of minimising errors on the design of the application itself.

2. Verification conditions can be used to enforce all functions calls satisfy the contract imposed by function specifications, implicitly validating Directive 4.11, which we can expend to any function and not only functions of the standard C library.

   To illustrate this, assuming the `dot` function starts with `require(a.length == b.length)`, the following code

   ```
   val a = Array(0, 1)     // someone forgot the third dimension
   val b = Array(1, 0, 0)
   dot(a, b)               // precond. (call dot(a, b)) is invalid
   ```

   results in an invalid precondition for the function call.

3. Rule 1.3 complements Directive 4.1 by requiring that there shall be no occurrence of undefined or even unspecified behaviour. *GenC* can ensure this partially by generating code that is not based, for example, on a particular order of evaluation of function arguments. But also by ensuring, through static verification, that no division by zero can occur, for example. Similarly to the previous rule, Rule 12.2 requires specifically that all operands of shift expressions are within valid ranges. And, Rule 18.1 completes Directive 4.1 by asking developers to pay special attention to invalid pointer arithmetic, which includes array access in C.

4. As detailed in the previous section the *mandatory* Rule 17.4 is ensured when all execution paths within a function are terminated by a **return** statement. Here too verification is highly convenient as illustrated by Listing 20.

```
def foobar(opt: Option[Int]): Int = opt match {
  case Some(x) if x >= 0 => x
  case None()            => 0
  // match exhaustiveness:
  //   The following inputs violate the VC
  //   opt := Some[Int](-2147483648)
}
```

Listing 20: Code exhibiting potential `MatchError`.

8.4 Code Generator Guarantees Enforcing Many Rules

As we just shown, static analysis can be beneficial to spot where and when a piece of code can fail to abide to a given set of rules. But a larger chunk of the MISRA Guidelines can be enforced by carefully generating valid C99 code. We discuss here the *required* directives and rules that *GenC* always respects.

1. Directive 2.1 and Rule 1.1 require that the generated code must compile without errors and contain no transgressions of the standard C syntax while respecting the limitations of the used C compiler. As discussed above, *GenC* assumes a compliant C99 compiler is available for compilation and therefore can uses the whole spectrum of the standard C syntax, without additional limits. It is also careful to generate exclusively standard C99 code, without embedding assembly instructions.

2. As discussed through this report, *GenC* uses exclusively stack allocated objects. It directly follows that Directive 4.12, which asks that no dynamic allocations are used, is fulfilled.

3. Rules 3.1 and 3.2 are both about comments within C code, adding some requirements on the syntax defined in the C standard. Because comments are solely addressed to human being they are completely irrelevant at runtime. The syntax of comments in C and Scala is also not quite identical; e.g. the latter allows nested comment blocks. Mainly for these two reasons *GenC* does not translate Scala comments into the generated C code. Additionally, comments are used only for readability purposes, delimiting several parts of the code such as `include`s or function declarations. To that end we exclusively use the `/∗block syntax ∗/`.

4. Rules 5.2, 5.3 and 5.7, which regulate usage of identifiers, should be enforced by the user as we discuss in Section 8.6 but *GenC* does ensure both Rule 5.5 is respected by not using macros. Rule 5.6 is also respected because all identifiers used in **typedef**s are unique, thanks to the extra unique postfixes added by *GenC*.

5. Regarding declarations and definitions, the guidelines specify that types should be explicitly specified to avoid using the implicit **int** type, prototype forms of functions should always have named parameters and all declarations must be consistent with name and

type qualifiers usage. For *GenC* it is trivial to produce code respecting those three points, detailed by Rules 8.1, 8.2 and 8.3: C programs need only to be rigorously constructed.

6. Because programs generated by *GenC* all consist of one compilation unit, **`static`** storage class is used for all functions, except `main` for which we use the standard signature. Furthermore, no global variables are allowed. Hence, Rule 8.8 is respected.

7. Rule 9.4, which requires that array elements or structure attributes must be initialised at most once, is automatically met because Scala already ensures this behaviour by initialising any object exactly once and because *GenC* map one-to-one an object initialisation.

8. As mentioned in Section 6.1 *GenC* ensures the evaluation model of Scala is respected by the generated code. This model is strict enough to determine a specific order of expression evaluation, such as evaluating function arguments or operands from left to right. The C standard however leaves many details up to the C implementation. For that reason, Rule 13.2 wants that all possible evaluation orders result in the same side-effects. It therefore follows that by enforcing a unique evaluation order the generated code abides to that rule.

9. Because it is unsafe to keep using the address of an object after it is released, *GenC* never returns from function a pointer to an object with automatic storage. Only pointer to string literals, which have a static storage class, can be returned. References are only used when passing a mutable object to a function to propagate side-effects to the caller site. Furthermore, because no aliasing is allowed, it is not possible to keep a reference to an object declared in an **`if`**, **`while`** or block expression beyond its scope. The logical conclusion is that the transpiled Scala code always follows Rule 18.6, which specifically requires that the address of an object x with automatic storage is never stored in an second object y outliving x.

10. When using **`#include`** directives, *GenC* exclusively uses the `<filename>` notation with headers from the Standard C Library, de facto ensuring both Rule 20.2 and 20.3 are respected. Moreover, this is the only usage of the # symbol, making all its usages valid preprocessing directives as requested by Rule 20.13.

11. The MISRA documentation acknowledges that the type system used in C is far from trivial and defines the *essential type model* to help developers understand and control implicit and explicit conversions through portable coding practice. It goes further than the ISO C Standard and defines several categories of *essential types*, on top of which Rules 10.1 to 10.8 are built to guide programmers into using the proper types for the various arithmetic operations.

    Many of these rules are automatically respected by *GenC*, mainly because we generate code that follows the strict Java specification which already imposes that all arithmetic operations are carried on 32-bit integers. And, as discussed in Sections 6.2 and 6.3, we carefully craft C code to use safe shift expressions, respecting instructions of Rule 10.1.

8.5 Deviations from the Guidelines

GenC do not abide to 5 rules from the MISRA Guidelines: 2 *readability* rules, 2 *advisory* rules and 1 *required* rule. We detail here the reasons why we either ignore or deviate from these and why it does not impact the safety of the software.

Readability **Rules**

1. Directive 4.5 recommends using typographically unambiguous identifiers. While this is partly up to the user, *GenC* also generates additional variables and rename all identifiers to be unique. In doing so, it could happen that identifiers can be said to be typographically ambiguous under this directive. This however does not affect the execution of the program and therefore the generated program is as safe as the input program in that respect.

2. Rule 2.7 asks that no function parameter should be unused. First of all, Scala and Leon do not forbid having unused parameters and therefore the user is accountable for providing code free of unused function parameters. That being said, due to the way nested functions and their respective context are translated into top-level functions *GenC* could generate functions with unused parameters. Since the user is authorised to provide a custom implementation for any function, *GenC* is not capable of computing the set of used variables exclusively from the Scala code.

   We understand the rationale behind this rule, which desire is to avoid mismatch between specification and implementation, but do note that regarding performance this is not an issue as all functions have internal linkage, therefore allowing a C optimiser to remove any extra arguments.

Advisory **Rules**

1. Rule 8.13 advises using pointers to `const`-qualified types whenever possible. The produced code however does not attempt to fulfil this rule nor does it use other `const`-qualified types because this concept does not exists in Scala: types are either mutable or immutable. It follows that this rule is not relevant for *GenC*.

2. Rules 19.2 claims that `union` types should not be used as they are deemed too complex and error-prone, putting at risk the security of the software. With *GenC* we carefully avoid any illegal instruction and keep usage of `union` to the minimum as detailed in Section 5.3, producing what we strongly believe to be safe C programs.

Required **Rule**

1. Rule 7.4 requires that string literals are only assigned to pointers to `const`-qualified `char`. The reasons why *GenC* uses `char*` and not `char const*` are twofold. On the one hand, the C99 standard defines the type of a string literal as `char [N]`, with `N` being the size of the string plus one to accommodate for the implicit sentinel character, which decays to `char*`. It also clearly specifies that mutating such an array results in undefined-behaviour but it remains that `char*` is a valid type to represent string literals. On the other, because *GenC* does not yet provide support for string manipulation, it is impossible to execute illegal instructions related to string literals, which is the rationale behind Rule 7.4. In that sense, the rule is fulfilled.

   We do however note that this model should be reconsidered when implementing a wider support for string manipulations for *GenC*.

8.6 Responsibilities Left to the User

GenC tries to ensure as many rules as possible are met by generating appropriate programs. For a considerably large part of the Guidelines it is capable of doing that, as we have discussed above. For some rules however *GenC* needs to collaborate with the user, and for some other rules *GenC* has to trust that the user provides Scala code that observes the mentioned directives and rules.

There are 16 *required* and 4 *advisory* rules that *GenC* cannot deal with and therefore are the sole responsibility of the user. We detail those now in an order of what we believe to be their relative importance:

1. Directive 3.1 requires that all code shall be traceable to documented requirements. It follows that to claim MISRA compliance one has to follow the process described in the guidelines for Scala code as well. Because the generated code uses identifiers close to the input program one can easily link documentation for a produced C function, type or variable back to the original function, type or variable.

2. Directive 4.7 mandates that error information retuned from function should be tested. To that end we highly recommend users to use, for example, the monadic `Option` class to represent the absence of results as one has to test the type of the returned instance, using pattern matching for example, to extract results and therefore make testing for errors unavoidable.

3. Rule 17.7 asks that every value returned by a function should be used, except for the **`Unit`** value, of course.

4. Rules 2.1 and 2.2 demand that the source code should not contain unreachable or dead code. The distinction between the two is that, in MISRA terminology, dead code is code that can be executed but does not affect the outcome of the program while unreachable code is code that is never reached, under any input. *GenC* is designed to not add such undesirable instructions but cannot remove such code. Moreover, we highlight the fact that such constructs can be the symptoms of design issues or bugs.

5. Rule 17.2 reminds the user that recursive functions, either by directly calling themselves or through intermediate function calls, can result in stack space shortage and therefore is unsafe. We discuss this particular issue in Section 5.4.

6. Rule 18.8 strongly suggests avoiding using *Variable Length Array* (VLA). *GenC* allows the user to have VLA but will warn the user. The user must ensure that the runtime size of such array is strictly positive to avoid undefined-behaviour.

7. Rules 13.1 and 13.5 both require the code to avoid side-effect in specific contexts; the former when initialising objects or arrays, the latter in operands of the logical && and || operators. By extension to Scala we recommend users to avoid side effect in any operator operands and function arguments, although the generated code ensures through normalisation that such instructions have the same behaviour in C.

8. Rule 14.3 puts more weight on the fact that loops should progress, and therefore have non-constant controlling expressions that are not invariant, in order to avoid undefined-behaviour and the nasal demons it nourishes [30]. Exceptions to this rule allow *constant expressions*[4] to produce infinite loops such as `while (true) { /*... */}`.

9. *GenC* produces code free of variable shadowing by means of identifier postfixes but the input code should be free of such issue in the first place to truly respect Rules 5.2 and 5.3.

[4]A *constant expression* can be evaluated during translation rather than runtime; for the full definition refer to Section 6.6 *Constant expressions* of the C99 Standard.

Similarly, Rule 5.7 strongly advises against reusing the same identifier for two different types. Since C99 does not support function overloading, it could be argued that this rule also applies to functions. Even though the generated code is technically free of such issue due to the disambiguating used postfixes, we recommend users to follow this guideline in their Scala code for readability purposes

10. Rule 22.3 asks that no file should be read and written at the same time through different streams. The provided I/O API cannot prevent this.

11. Rule 21.6 explains why one should not use features from the C Standard Library I/O framework. We let the user determine whether this restriction applies to their project or not. If I/O through `FILE*` is problematic the user should not use features from the `leon.io` package.

12. Directive 4.13 advises to release resources in the opposite order they were acquired. As it currently stands, only `FileInputStream` and `FileOutputStream` require to be released using their respective `close()` method, but the user is free to create other proxies for system resources using annotations presented in Section 7.

13. Rule 8.9 recommends to define variables, and by extension to Scala nested functions as well, the closest to their usage as possible.

14. Rule 12.4 advocates against having (unsigned) wrap around evaluation in *constant expressions* to avoid confusion. Note that when processing the input program such expressions are constant folded to produce the final value automatically[5]. This means that expressions such as `val x = 2147483647 + 1` are translated into `int32_t x = -2147483648;`.

15. Rule 15.7 requires every `if ... else if` constructs to followed by a final `else` expression, possibly only consisting in a comment explaining why no code is executed in this final branch. Scala do force `if` statements to have a final `else` branch but only when they are used to "return" a value. In other words, examples such as the following one are valid code but do not respect this rule.

```scala
def foo(b: Boolean): Int = {
  if (b) log("foo") // no explicit else
  42
}
```

16. Rule 15.5 advises to end functions with a unique `return` statement. In the supported fragment of Scala, all functions end with a unique expression as the `return` keyword is not allowed. However, *GenC* inserts more that one `return` statements in a function when it is ended by either an if-else expression or a pattern matching expression. To follow this rule, one can always manually create a temporary variable for such expressions and then return the variable instead, as we illustrate in Listing 21.

Finally, when using annotations presented in Section 7 in order to use existing C libraries, users should be aware that they are injecting C code not handled by *GenC*. Therefore, users should double check they do not violate *any* rule.

[5]This happens in Leon when extracting the AST using `scalac`.

```scala
def invalid1(opt: Option[Int]) = opt match {
  case Some(42) => true
  case _        => false
}

def valid1(opt: Option[Int]) = {
  val res = opt match {
    case Some(42) => true
    case _        => false
  }
  res // ensures only one ‘return‘
}

def invalid2(x: Int) = if (x < 0) x else -x

def valid2(x: Int) = {
  val res = if (x < 0) x else -x
  res // ensures only one ‘return‘
}
```

Listing 21: Scala functions validity with respect to Rule 15.5.

9 Case Study: LZW

To showcase the abilities of *GenC* we implemented the well-known Lempel–Ziv–Welch (LZW) compression/decompression algorithm [9]. After laying down the weft of the algorithm we define how strings are represented. Then we discuss the structure of two implementations in Scala. And finally we give some benchmark measurements of the compression ratios and comment on the speeds of the encoding & decoding processes.

9.1 The LZW Algorithm

In LZW a coding table, or *dictionary*, representing a mapping between known pattern and codewords is constructed on the fly as the data to encode or decode is read. The way this dictionary is built requires that the encoder and decoder only share one information beforehand: the supported and ordered input alphabet. It allows the decoder to reform the original data without being provided with the full encoding dictionary.

In this report we discuss one of the simplest versions published in the 80's which was used in several tools. Improved versions are still used in, for example, the GIF format. Compared to other compression algorithm its compression ratio makes it really attractive while keeping a relatively simple structure.

The compression, decompression and helper procedures for the algorithm are exposed in Listing 22. Errors, such as premature end of file, are left out here for simplicity but our implementations do handle them.

Having an initial dictionary of codewords mapping to the default alphabet allows the encoding and decoding procedures to always be able to handle new inputs: in the worst case scenario when no pattern repetition occurs only strings of length one are processed.

The algorithm compression power is however limited by the size of the dictionary: the dictionary is considered full when all codewords have been associated with a string. In that situation several alternative are possible. One could always stop the encoding or decoding phase and report an error. Because this is not satisfactory in out opinion, we instead decided to enter an alternative encoding/decoding mode in which only existing codeword are used and the dictionary stop growing. This allows processing files of all sizes. Other options includes clearing the dictionary when full, except for the codewords for the base alphabet, or removing the oldest/less frequently used codeword when a new one is needed.

9.2 Implementation: String Representation

Since no heap allocation is allowed we encoded strings using a fixed-length buffer of characters, represented by the class `Buffer` which is sketched in the following listing. We also require that all buffer instances have a `capacity` of `BufferSize`, which is a global parameter of the program, by means of verification conditions that is proven to be correct. This requirement is helpful to simplify reasoning about `Buffer`. The actual implementation, which is available in Appendix B, has additional methods that are only relevant for the implementation details of the algorithm.

The characters themselves are, for the purpose of this benchmark, **`Byte`**; this allows us to use a `leon.io.FileInputStream` and read file byte by byte, without having to interpret the content of files nor care about encoding when the contents are text-based.

This string representation allows us to append characters, access individual characters, and more, but under the strict restrictions that the capacity of the buffer is not exceeded and all array accesses are valid. Such safety properties are verified statically and therefore entitles the program to assume all array accesses are in the allocated bounds.

```
PROCEDURE initialise dictionary, INPUT dictionary D:
  for each character c in the alphabet
    associate c with the next available codeword and
    insert the relation in D
end

PROCEDURE encode, INPUT string data:
  let D be an empty dictionary
  initialise dictionary D

  w <- "" (the empty string)

  for each character c in data:
    wc <- concatenate w and c
    if D contains wc:
      w <- wc
    else:
      read the codeword o in D for w
      output o
      associate cw to the next free codeword and
      add the relation in D
      assign c to w
    end
  end

  read the codeword o in D for w
  output o
end

PROCEDURE decode, INPUT sequence of codewords data:
  let D be an empty dictionary
  initialise dictionary D

  previous <- consume head of data
  read the string o associated with p in D
  output o

  for each codeword current in data
    read the string s associated with current in D
    output s

    let c be the first character of s
    read the string t associate with previous in D
    wc <- t + c
    associate wc to the next free codeword and
    add the relation in D

    previous <- current
  end
end
```

Listing 22: The three main procedures of the LZW algorithm.

We show here the main component of this `Buffer` class:

```
case class Buffer(private val array: Array[Byte], private var length: Int) {
  val capacity = array.length
  def size = length

  def isFull: Boolean = length == capacity
  def nonEmpty: Boolean = length > 0

  def isEqual(b: Buffer): Boolean = // ...
  def apply(index: Int): Byte = // ...
  def append(x: Byte): Unit = // ...
}
```

9.3 Implementation: Dictionary Representations

The two implementation variants we present here use two different representations for the dictionary. Both are using a linear search algorithm to find a key-value pair in an array but using distinct memory layouts and encodings for the codeword-string pairs. We detail in this section how the two versions differ but also what is their common denominator.

As implicitly mentioned above the input alphabet is the range covered by **`Byte`**, i.e. $[-128, 127]$, and this is true for all dictionary variants. Additionally, both use fix-sized, 16-bit codewords similarly as the initial version of LZW. We were also able to use the same public interface for both, which spare us the need of rewriting the encoding and decoding functions:

```
case class Dictionary(private /* implementation-defined */) {
  def nonFull: Boolean
  def lastIndex: Int // requires non-empty dictionary
  def contains(index: Int): Boolean
  def insert(b: Buffer): Unit // requires non-full dictionary
  def encode(b: Buffer): Option[CodeWord]
}
```

Dictionary Design A: Array of `Buffer`

```
case class Dictionary(
  private val buffers: Array[Buffer],
  private var nextIndex: Int
) { /* ... */ }
```

As shown above, the first implementation of the dictionary is based on an `Array[Buffer]`: the values (string buffers) are simply the values stored in the array while the keys (codewords) are the indexes. This array is encapsulated in a `Dictionary` case class, which also keeps track of the next free place in the array. Because indexes are used to represent 16-bit codewords the underlying array cannot contain more than 2^{16} elements or some indexes could not be represented as codewords without being ambiguous.

When using the dictionary to encode data the algorithm needs to find out whether the current candidate string (`wc` in the pseudo-code above) is in the dictionary. To do that, the dictionary array is traversed from its first element to its last used element in sequential order. Because codeword-string pairs do not need to be removed from the dictionary in LZW all valid buffers are stored to be contiguous in the array. It follows that the complexity for looking up the potential codeword, or its absence, for a given string is $\mathcal{O}(n \times \texttt{BufferSize})$ where n is the size of the dictionary and `BufferSize` takes into account the complexity of comparing two string buffers.

While this runtime complexity is not optimal, given a codeword as in the decoding procedure we can get the corresponding string in $\mathcal{O}(1)$ using a random access lookup in the dictionary array.

This implementation works decently at runtime, considering that it performs a linear lookup, but suffers from mainly two tangled issues. To begin with, because only static allocation is allowed with *GenC*, each array needs to be associated with a variable in the generated code in order to be kept alive. And since the size of the array is not present in the array type, as discussed in Section 5.2, *GenC* does not include this information in the type system. For example, the `Buffer` class is represented as follows:

```
typedef struct { int8_t* data; int32_t length; } array_int8;
typedef struct { array_int8 array; int32_t length; } Buffer;
```

Because of this indirection, arrays such as `array_int8` do not directly own the memory allocated for the array. By transitivity, an instance of the `Buffer` structure does not directly own all the memory associated with it. To illustrate this,

```
val b = Buffer(Array.fill(64)(0), 0)
```

is translated into the following C99 code[6]:

```
int8_t leon_buffer_0[64] = { 0 };
array_int8 norm_0 = (array_int8) { .data = leon_buffer_0, .length = 64 };
Buffer b = (Buffer) { .array = norm_0, .length = 0 };
```

Because *GenC* enforces that `b` does not outlive `leon_buffer_0` it is safe to assume the memory for the array of characters will always be available. However, the translated code can *degenerate* when nested arrays are involved. This is the case with the implementation of `Dictionary` defined earlier. The C code for

```
val d = Dictionary(Array.fill(1024){ createBuffer() }, 0)
```

where `createBuffer()` is an inlined function that returns `Buffer(Array.fill(64)(0), 0)`, is reported in Listing 23.

The second issue is a consequence of the number of variables and nested arrays that grows fast with the size of arrays: it creates significantly large programs in terms of line of code. Moreover, *GenC* can be noticeably slow to generate such programs, the compile time, especially with optimisation enabled, takes a considerable amount of time and the generated assembly code will be predictably huge.

[6]The presented code is slightly simplified for the sake of this example; the actually produced code would be a bit more complex to accommodate for some general normalisation transformations that are not relevant here.

```
typedef struct { Buffer* data; int32_t length; } array_Buffer;
typedef struct { array_Buffer buffers; int32_t nextIndex; } Dictionary;

// In a function:
int8_t leon_buffer_0[64] = { 0 };
array_int8 norm_0 = (array_int8) { .data = leon_buffer_0, .length = 64 };
Buffer norm_1 = (Buffer) { .array = norm_0, .length = 0 };

int8_t leon_buffer_1[64] = { 0 };
array_int8 norm_2 = (array_int8) { .data = leon_buffer_1, .length = 64 };
Buffer norm_3 = (Buffer) { .array = norm_2, .length = 0 };

// ...

int8_t leon_buffer_1023[64] = { 0 };
array_int8 norm_2046 =
    (array_int8) { .data = leon_buffer_1023, .length = 64 };
Buffer norm_2047 = (Buffer) { .array = norm_2046, .length = 0 };

Buffer leon_buffer_1024[1024] = { norm_1, norm_3, /* ... */, norm_2047 };
array_Buffer norm_2048 =
    (array_Buffer) { .data = leon_buffer_1024, .length = 1024 };

Dictionary d = (Dictionary) { .buffers = norm_2048, .nextIndex = 0 };
```

Listing 23: Code generated with **Design A** for
`val d = Dictionary(Array.fill(1024){ createBuffer() }, 0)`

Dictionary Design B: Custom & Compact Memory Layout

```
case class Dictionary(
  private val memory: Array[Byte],
  private val pteps: Array[Int],
  private var nextIndex: Int
) { /* ... */ }
```

With the second design of `Dictionary` we try to address the two issues related to the size of the generated code. It is also an attempt at squeeze more strings into the same amount of memory.

By analysing statistics of memory usage of the previous implementation, we notice that the vast majority of string buffers were not using their full capacity at all; instead most of them are short. For example, all the codewords for the initial alphabet are associated with strings for length one, effectively wasting `BufferSize - 1` bytes of memory.

Consequently, we represented a `Dictionary` storage using a raw memory region `memory`, allocated on the stack using an `Array[Byte]`, and an array of integers `pteps`[7], holding for each string in the dictionary the index in `memory` of the next `Byte` after that string. A dictionary can

[7]`pteps` stands for *Past The End PointerS*.

therefore contain at most `pteps.length` strings, assuming `memory` is big enough to hold all of them. The `Dictionary` case class also has an attribute keeping track of the next free index (and therefore codeword).

It is arguably not obvious but `pteps` can be used to compute the size of each string stored in `memory`: the length of the first string is `pteps(0)` itself while the *i*-th string has a length of `pteps(i) - pteps(i-1)`.

As illustrated Appendix B.1, this new design prohibits us from reusing some part of `Buffer`, such as `Buffer.set` to insert new elements in the dictionary, and requires us to write more complex code even for simple functions such as `nonFull`. It also follows that the verification conditions used to prove the correctness of this class are notably more involved.

Regarding the generated code, creating a dictionary such as

```
val d = Dictionary(Array.fill(524288)(0), Array.fill(8192)(0), 0)
```

results in tremendously smaller code size as reported in Listing 24.

```
typedef struct { array_int8 memory; array_int32 pteps; int32_t nextIndex; }
       Dictionary;

// In a function:
int8_t leon_buffer_0[524288] = { 0 };
array_int8 norm_0 = (array_int8) { .data = leon_buffer_0, .length = 524288 };
int32_t leon_buffer_1[8192] = { 0 };
array_int32 norm_1 = (array_int32) { .data = leon_buffer_1, .length = 8192 };
Dictionary* d =
    (Dictionary) { .memory = norm_0, .pteps = norm_1, .nextIndex = 0 };
```

Listing 24: Code generated with **Design B** for
`val d = Dictionary(Array.fill(524288)(0), Array.fill(8192)(0), 0)`

The direct consequence of this is that *GenC* is capable of generating C99 for this implementation quickly, and C compilers are able to compile the program in a breeze, even with aggressive optimisation turned on, and produce a relatively small assembly code. We will see next that this second design compresses files with a smaller memory footprint but unfortunately not as quickly as the previous implementation.

9.4 Benchmark Results

The first remark we can make about this experiment is that *GenC* is mature enough to transpile a Scala implementation of the famous LZW algorithm. It is true that when dealing with some forms of array the generated code can be large and therefore slow to produce and compile, but we also showed how to avoid this issue at the cost of scarifying the *Don't Repeat Yourself* (DRY) idiom. By customising the implementation of `Dictionary` as we have shown with the **Design B** that we were also able to reduce the memory footprint of the program at runtime and the size of the binary produced by Clang, as reported in Table 1.

We used commands similar to the following to transpile the LZW implementations, compile them using the Clang C compiler and execute them:

Transpiling `time leon --genc testcases/genc/LZWa.scala --o=LZWa.c`

Compiling `time clang LZWa.c -o LZWa -O3 -std=c99 -DNDEBUG`

Running `time -l ./LZWa`

Time measurements for the runtime of both C99 implementations were done 10 times and are reported as an average in Table 1. All commands were run on a *MacBookPro10,1* with a quad-core 2.6Ghz Intel Core i7, 16GB of main memory, 6MB of L3 cache and 256KB of L2 cache per core, running macOS 10.11.6. The version of Clang used for these experiments is *Apple LLVM version 8.0.0 (clang-800.0.42.1)*.

A file of 755KB was used for this benchmark; all implementations were able to compress and decompress it successfully, yielding exactly the original data. Using an implementation with heap allocation written in C++ we were able to determine that, for our input data, a total of 55850 codewords were needed to optimally compress it down to 6.8% of its original size, the longest string stored in the dictionary having a length of 65. We used for **Design A** a `DictionarySize` of 8192 and a `BufferSize` of 64. And, for **Design B** we additionally set `DictionaryMemorySize` to 524288, i.e. `DictionarySize * BufferSize`.

These results show that **Design A** is about 16 times slower to transpile compared to **Design B**, with a C99 code close to 200 times bigger in size. Not surprisingly, compiling the first implementation using most of Clang optimisations is roughly 1500-2000 time slower than its alternative implementation, with a binary size ∼80 times bigger. However, contrary to the intuition that less assembly instructions is better for speed, the initial version is 150% faster, using the wall clock to measure time, than the second one when both are tasked to compress the same file with the same maximum number of codewords to achieve a compression ratio for 23.2%. Finally, we can remark that **Design A** uses twice as much resident memory than **Design B** at their peak.

	Design A	Design B	Comment
Transpiling			
	∼200s	∼12s	wall clock time
	5.3MB	28KB	size of C99 code
Compiling			
	∼650s	<1s	wall clock time
	1.0MB	13KB	size of assembly code
Runtime			
	6.64s	9.91s	average wall clock time
	6.60s	9.85s	average user time
	0.02s	0.01s	average system time
	2408KB	1256KB	maximum resident set size
	23.2%	23.2%	compression ratio

Table 1: Measurements for LZW implementations

Because this benchmark does a lot of input and output operation using the filesystem, it is important to study whether the running times measured are significantly impacted or not by blocking kernel operations. As it turns out, using the POSIX `time` utility we can see by looking at the difference of clock measurements (*real* or *wall*, *user* and *system*) that blocking operations account for a marginal portion of runtime. Moreover, this utility reported that no page fault occurred. It is therefore highly likely that the operating system cached the whole input files in main memory and optimised the output operations as well.

Interestingly, by looking at a breakdown of the CPU usage per function for both implementations, generated by profiling them using Apple's *Instruments* tool and Clang compilation options `-O3 -std=c99 -fno-omit-frame-pointer -g -DNDEBUG -fno-inline-functions`, we can make two conclusions:

1. The decoding process, regardless of the implementations, only accounts for less than 1% of the total runtime. This is obviously because looking up codewords in the `Dictionary` is $\mathcal{O}(1)$ and therefore very efficient.
2. In both cases, 50% or more of the time is spent comparing string buffers. As previously noted, looking up for string in the dictionary is $\mathcal{O}(n \times BufferSize)$, where n is the size of the dictionary, and this is the bottleneck of the algorithm. It therefore would be wise to use a different data structure, for example based on hash functions, to represent the dictionary and improve the runtime performance of the programs.

Finally, because the Scala code used to implement both designs is valid, we are able to run both versions on the JVM. The compilation process lasted for about 15 seconds for both programs, but their running time were 160s and 200s, respectively. Contrary to the C equivalent code, these programs run all assertions at runtime and both spend more than 50s in kernel functions, as reported by `time` and use much more memory. Furthermore, the difference of runtime efficiency is probably explainable by common issues of software running on the JVM, such as autoboxing [31] or possibly an inefficient implementation of `Array[Byte]` compared to low level `int8_t[]` in C. With that in mind it would not be fair to compare the performance of the JVM-based implementations to the transpiled programs as no effort was made to avoid these well-known limitations.

10 Conclusion

The goals achieved in this project are multiple. First, we introduced the ability to use **`Byte`** with Leon with the proper integer promotion strategy used by Java and Scala. We also updated Leon to add automatic verification for a wider range of potential integer overflows and to detect unexpected behaviour in arithmetic expressions. Additionally, we introduced an Input/Output API for standard streams and files, with a backend for C99.

After carefully analysis the requirement imposed by the *xlang* module regarding aliasing we devised a stack-allocation based memory model, free of dynamic allocation, that is capable of handling most form of mutability in class hierarchies, and therefore imposes no requirement on heap memory management.

Not only did we introduce support for **`Byte`** and other primitive types, but we managed to transpile generic types and functions into C code without using language extensions nor introduce a runtime performance penalty. Next, we presented a stronger model for normalisation of Scala code in order to properly project the evaluation semantics of the input software onto the generate one. We also significantly improved the safety of shift operations, thanks to a close inspection of the Java 8 and C99 standards to understand exactly when an expression is equivalent to another in both languages and, especially for C, when undefined-behaviour can be triggered. With a better comprehension of the limits of C programs we were able to use different integer types to workaround some dangerous pitfalls by converting values from one to another type.

Furthermore, support for casts and membership tests within class hierarchies were added as well as the ability to convert pattern matching expressions into equivalent, imperative code. And in order to use stable and trusted C libraries we designed an annotation systems to create proxy interface, providing a bridge between Scala and C code.

One central aspect of this report is that we documented how *GenC* is capable of respecting, with very few exceptions, the strict MISRA Guidelines for critical and embedded systems development, exposing when the verification tools of Leon can be extremely beneficial and what the user is required to do to work his way through full compliance.

We ended the discussion by using the LZW data compression and decompression algorithm to illustrate how successfully *GenC* can perform on real-life programs, although identifying some limitations with nested arrays that are the consequences of our memory model.

11 Future Works

We end the present document by opening the discussion on several points that would benefit from further development:

- Integrating optimisations in the generate code could be done by performing some analysis on the AST to determine whether side-effect is present or not in a given set of expressions; if expressions are all pure the normalisation introduced in Section 6.1 could be avoided, generating simpler programs that C compilers could optimise more easily. Additionally, some **`if`** statements could be transformed into **`switch`** statements when, for example, the branching only depends on an **`enum`** value.

  Performance benefit from such optimisations are however hard to estimate because nowadays C compilers do already perform some aggressive optimisations when asked to.

- Resource inference introduced by Orb [26] to deal more efficiently with imperative programs, giving developers a good analysis of the memory requirement for their generated C99 programs, but also in term of speed were an accurate execution model be elaborated for a given hardware system.

- Adding an extra layer to the I/O library component of Leon to automatically close file handle when they go out of scope. This could be implemented with an inlined and curried function, similar to the one exposed below, fundamentally using the same trick than for `Option` discussed in Section 7.

```
@inline
def withFile(filename: String)
            (onSuccess: FileInputStream => Unit): Boolean = {
  val fis = FileInputStream.open(filename)
  if (fis.isOpen) {
    onSuccess(fis)
    fis.close()
    true
  } else false
}
```

  Alternatively, two callback functions could be passed to `withFile` and a generic return type could be inferred from them. But in any case the passed functions could only be anonymous lambdas as higher order functions remains a complex concept when expressed in C, especially while maintaining the semantics of Scala.

- With upcoming updates to the xlang module [8], the aliasing restrictions are expected to be slightly relaxed, essentially allowing a variable, or one of its member, to be aliased by another if only one of them is used subsequently.

```
def foo(x: MutableType) = {
  val y = x // Currently illegal but planned to be valid
  bar(y)    // as x is not used after y is created.
}
```

Such modification of the anti-aliasing rules would require a reassessment of how *GenC* handle mutability to determine whether the translation process remains sound.

◇ The memory model presented in Section 4 is known to suffer from one flaw illustrated in Listing 8. Two opposite updates are interesting at this point. The first would be to notify the user when *GenC* detects code that it cannot properly handle, possibly stopping the translation process.

  The other alternative would be to considerably change the ownership model used by *GenC* to represent types in two fashions: *owner* and *reference*. The former would have exclusively value attributes and own them. The latter would be bound to an *owner* and have references for its mutable fields. Additionally, we should be able to convert an *owner* to a *reference* but not vice-versa. This framework allows passing mutable tuples or case classes to functions and respect aliasing (and therefore propagates mutation) when functions can return only the *owner* variant of a type.

  Applied to Listing 8, instead of producing Listing 10 *GenC* could generate a program along these lines:

```
typedef struct { int x; } A;
typedef struct { A   a; } B_owner;
typedef struct { A*  a; } B_ref;

static void update(A* a) {
  updateB((B_ref) { .a = a });
  /* no copy:            ^  */
}
```

  Such modification of the type translation and representation however would require a full analysis, assessing notably if the augmentation of types and type conversions impacts the generated C99 code in an undesirable way. Moreover, the impact of the relaxed aliasing rules described above should be taken into account when solving this limitation.

◇ In order to make the process to have MISRA compliant C code developers using *GenC* need to pay attention to several points discussed in Section 8.6. Documenting and keeping track of how each point is taken care of is obviously a tedious task when done solely manually. However, for several points Leon and *GenC* could emit warnings and instruct users how to solve specific issues. We list here several points of interest for a linter-like report:

  1. Directive 4.7 and Rule 17.7 both ask that value returned by function should be used – the former specifically address error values signalled by functions. Automatically reporting when a value is not read should be possible locally to each function body.
  2. Identifying systematically unreachable or dead code would directly help fulfilling Rules 2.1 and 2.2.
  3. Side effect present when initialising arrays, objects or in operands would make it trivial to check for compliance to Rules 13.1 and 13.5.
  4. Similarly, the benefits of reporting places in the user code where an identifier shadows another one are twofold: it would make Rules 5.2, 5.3 and 5.7 obsolete and this would allow *GenC* to avoid using complex postfixes on identifiers in many places, consequently making the generated C code looking more similar to the original Scala code.

5. Using simple checks on the AST Leon and *GenC* could determine whether all **if** statements have an **else** branch in order to satisfy Rule 15.7. Similarly, when multiple **return** expressions are inserted by *GenC* a warning could be emitted for Rule 15.5.
6. Finally, having the ability to detect overflows in *constant expression* within Leon would allow use to warn the user when Rule 12.4 is not respected.

A Operator Equivalence Table

Here we briefly present an analysis of the different arithmetic operators present in Java and C, highlighting the differences of semantics and behaviour as well as identifying the legal range of values.

The first column determines whether the C99 operator presented in the second one takes one argument (*unary*) or two arguments (*binary*). The arguments are denoted by *rhs* and *lhs*, which stand for right- and left-hand side. The seventh column contains the closest Java operator. The valid range of values is detailed in columns four and five for the C99 standard and in columns nine and ten for the Java environment. We refer in columns three and eight to which sections of the C Standard [20] and Java Standard [21], respectively, define the exact semantic of the operators.

Then we highlight in the last column the main differences between the C and Java operators. And finally, in the sixth column we use four symbols that specify under which condition the C99 and Java semantics of the operator are equivalent. Their respective meaning is as follows when the operations are carried out on 32-bit integers:

✓
: The two semantics are completely equivalent.

$\mathcal{O}$
: The semantics are equivalent when no overflow occur.

$\mathcal{S}$
: Under `--strict-arithmetic` verification mode the two are identical.

×
: There is no direct equivalence between the operator.

Category	**C operator**	**C99**	**lhs range**	**rhs range**	**Semantic**	**Java operator**	**Java 8**	**lhs range**	**rhs range**	**Comment**
unary	`+`	6.5.3.3	any int	N/A	✓	`+`	15.15.3	any int	N/A	Not in Leon
unary	`-`	6.5.3.3	all but `INT_MIN`	N/A	$\mathcal{O}$	`-`	15.15.4	any int	N/A	In Java `-INT_MIN == INT_MIN`
binary	`*`	6.5.5	any int	any int	$\mathcal{O}$	`*`	15.17.1	any int	any int	
binary	`/`	6.5.5	any int	non-zero	$\mathcal{O}$	`/`	15.17.2	any int	non-zero	◇ C99 defines division in the same fashion as Java and C++11. ◇ In Java `INT_MIN / -1 == INT_MIN`. ◇ `INT_MIN / -1` overflows in C.
binary	`%`	6.5.5	any int	non-zero	$\mathcal{S}$	`%`	15.17.3	any int	non-zero	◇ `MIN_INT % -1` is undefined in C11. ◇ It is unclear in C99 [24]. ◇ In Java the result is defined to be 0.
binary	`+`	6.5.6	any int	any int	$\mathcal{O}$	`+`	15.18.2	any int	any int	
binary	`-`	6.5.6	any int	any int	$\mathcal{O}$	`-`	15.18.2	any int	any int	
unary	`~`	6.5.3.3	any int	N/A	✓	`~`	15.15.5	any int	N/A	
binary	`&`	6.5.10	any int	any int	✓	`&`	15.22.1	any int	any int	
binary	`^`	6.5.11	any int	any int	✓	`^`	15.22.1	any int	any int	
binary	`|`	6.5.12	any int	any int	✓	`|`	15.22.1	any int	any int	

Category	C operator	C99	lhs range	rhs range	Semantic	Java operator	Java 8	lhs range	rhs range	Comment
binary	signed `<<`	6.5.7	see comment	$[0, 31]$	$\mathcal{S}$	`<<`	15.19	any int	any int	◇ In Java, only the 5 lowest-order bits of rhs are used (range $[0, 31]$). Moreover, the bits are always shifted as one would expect. ◇ In C, assuming rhs is in $[0, 31]$, we also need to have $lhs * 2^{rhs}$ in $[0, 2^{31} - 1]$; otherwise the behaviour is undefined.
binary	unsigned `<<`	6.5.7	any unsigned int	$[0, 31]$	$\mathcal{S}$	`<<`	15.19	any int	any int	◇ Assuming rhs is in $[0, 31]$, the unsigned version of « in C99 is equivalent to Java's. ◇ However, converting back from unsigned integer is implementation defined!
binary	signed `>>`	6.5.7	any non-negative int	$[0, 31]$	$\times$	`>>` or `>>>`[8]	15.19	any int	any int	◇ In Java, only the 5 lowest-order bits of rhs are used (range $[0, 31]$). ◇ Moreover, in Java, `>>` will sign-extend lhs while `>>>` will zero-extend lhs. ◇ In C99, signed integer right shift is implementation defined for negative integers.
binary	unsigned `>>`	6.5.7	any unsigned int	$[0, 31]$	$\mathcal{S}$	`>>>`	15.19	any int	any int	◇ In Java, only the 5 lowest-order bits of rhs are used (range $[0, 31]$) and lhs will be zero-extended. ◇ In C99, assuming rhs is in $[0, 31]$, basically, lhs gets zero-extended.

[8]Depending on the sign of lhs.

B LZW Implementations

For completeness, we give print here our two implementations of LZW written in Scala as well as the C99 programs generated by *GenC*. The sources are also available at https://github.com/mantognini/GenC.

B.1 Scala Source

Design A

```scala
/* Copyright 2009-2016 EPFL, Lausanne */

import leon.lang._
import leon.proof._
import leon.annotation._

import leon.io.{
  FileInputStream => FIS,
  FileOutputStream => FOS,
  StdOut
}

object LZWa {

  // GENERAL NOTES
  // =============
  //
  // Encoding using fixed size of word;
  // Input alphabet is the ASCII range (0-255);
  // A word is made of 16 bits (instead of the classic 12-bit scenario, for
    ↪simplicity);
  // The dictionary is an array of Buffer where the index is the key;

  // We limit the size of the dictionary to an arbitrary size, less than or
    ↪equals to 2^16.
  @inline
  val DictionarySize = 8192

  // We use fix-sized buffers
  @inline
  val BufferSize = 64 // characters

  val AlphabetSize = Byte.MaxValue + -Byte.MinValue

  private def lemmaSize: Boolean = {
    DictionarySize >= AlphabetSize &&
    BufferSize > 0 &&
    AlphabetSize > 0 &&
    DictionarySize <= 65536 // Cannot encode more index using only 16-bit
      ↪codewords
  }.holds

  // Helper for range equality checking
  private def isRangeEqual(a: Array[Byte], b: Array[Byte], from: Int, to:
    ↪Int): Boolean = {
    require(0 <= from && from <= to && to < a.length && to < b.length)
    a(from) == b(from) && {
      if (from == to) true
      else isRangeEqual(a, b, from + 1, to)
    }
  }

  private def allValidBuffers(buffers: Array[Buffer]): Boolean = {
    def rec(from: Int): Boolean = {
      require(0 <= from && from <= buffers.length)
      if (from < buffers.length) buffers(from).isValid && rec(from + 1)
      else true
    }

    rec(0)
  }


  // A buffer representation using a fix-sized array for memory.
  //
  // NOTE Use ‘createBuffer()‘ to get a new buffer; don’t attempt to create
    ↪one yourself.
  case class Buffer(private val array: Array[Byte], private var length: Int)
    ↪ {
    val capacity = array.length
    require(isValid)

    def isValid: Boolean = length >= 0 && length <= capacity && capacity ==
      ↪BufferSize
```

```
def isFull: Boolean = length == capacity

def nonFull: Boolean = length < capacity

def isEmpty: Boolean = length == 0

def nonEmpty: Boolean = length > 0

def isEqual(b: Buffer): Boolean = {
  if (b.length != length) false
  else { isEmpty || isRangeEqual(array, b.array, 0, length - 1) }
}

def size = {
  length
} ensuring { res => 0 <= res && res <= capacity }

def apply(index: Int): Byte = {
  require(index >= 0 && index < length)
  array(index)
}

def append(x: Byte): Unit = {
  require(nonFull)

  array(length) = x

  length += 1
} ensuring { _ => isValid }

def dropLast(): Unit = {
  require(nonEmpty)

  length -= 1
} ensuring { _ => isValid }

def clear(): Unit = {
  length = 0
} ensuring { _ => isEmpty && isValid }

def set(b: Buffer): Unit = {
  if (b.isEmpty) clear
  else setImpl(b)
} ensuring { _ => b.isValid && isValid && isEqual(b) }

private def setImpl(b: Buffer): Unit = {
  require(b.nonEmpty)

  length = b.length

  var i = 0
  (while (i < length) {
    array(i) = b.array(i)
    i += 1
  }) invariant { // TIMEOUT
    0 <= i && i <= length &&
    // lengthCheckpoint == b.length && lengthCheckpoint == length && //
↪no mutation of the length
    isValid && nonEmpty &&
    length == b.length &&
    (i > 0 ==> isRangeEqual(array, b.array, 0, i - 1)) // avoid
↪OutOfBoundAccess
  }
} ensuring { _ => b.isValid && isValid && nonEmpty && isEqual(b) }

}

@inline // very important because we cannot return arrays
def createBuffer(): Buffer = {
  Buffer(Array.fill(BufferSize)(0), 0)
} ensuring { b => b.isEmpty && b.nonFull && b.isValid }


def tryReadNext(fis: FIS)(implicit state: leon.io.State): Option[Byte] = {
  require(fis.isOpen)
  fis.tryReadByte()
}

def writeCodeWord(fos: FOS, cw: CodeWord): Boolean = {
  require(fos.isOpen)
  fos.write(cw.b1) && fos.write(cw.b2)
}

def tryReadCodeWord(fis: FIS)(implicit state: leon.io.State): Option[
  ↪CodeWord] = {
  require(fis.isOpen)
  val b1Opt = fis.tryReadByte()
  val b2Opt = fis.tryReadByte()

  (b1Opt, b2Opt) match {
    case (Some(b1), Some(b2)) => Some(CodeWord(b1, b2))
    case _ => None()
  }
}

def writeBytes(fos: FOS, buffer: Buffer): Boolean = {
  require(fos.isOpen && buffer.nonEmpty)
  var success = true
  var i = 0

  val size = buffer.size

  (while (success && i < size) {
```

```scala
    success = fos.write(buffer(i))
    i += 1
  }) invariant {
    0 <= i && i <= size
  }

  success
}

case class CodeWord(b1: Byte, b2: Byte) // a 16-bit code word

def index2CodeWord(index: Int): CodeWord = {
  require(0 <= index && index < 65536) // unsigned index
  // Shift the index in the range [-32768, 32767] to make it signed
  val signed = index - 32768
  // Split it into two byte components
  val b2 = signed.toByte
  val b1 = (signed >>> 8).toByte
  CodeWord(b1, b2)
}

def codeWord2Index(cw: CodeWord): Int = {
  // When building the signed integer back, make sure to understand
   ↪integer
  // promotion with negative numbers: we need to avoid the sign extension
   ↪here.
  val signed = (cw.b1 << 8) | (0xff & cw.b2)
  signed + 32768
} ensuring { res => 0 <= res && res < 65536 }


case class Dictionary(private val buffers: Array[Buffer], private var
   ↪nextIndex: Int) {
  val capacity = buffers.length
  require(isValid)

  def isValid = 0 <= nextIndex && nextIndex <= capacity && capacity ==
   ↪DictionarySize && allValidBuffers(buffers)

  def isEmpty = nextIndex == 0

  def nonEmpty = !isEmpty

  def isFull = nextIndex == capacity

  def nonFull = nextIndex < capacity

  def lastIndex = {
    require(nonEmpty)
    nextIndex - 1
  } ensuring { res => 0 <= res && res < capacity }
```

```scala
def contains(index: Int): Boolean = {
  require(0 <= index)
  index < nextIndex
}

def appendTo(index: Int, buffer: Buffer): Boolean = {
  require(0 <= index && contains(index))

  val size = buffers(index).size

  assert(buffer.capacity == BufferSize)
  if (buffer.size < buffer.capacity - size) {
    assert(buffer.nonFull)

    var i = 0
    (while (i < size) {
      buffer.append(buffers(index)(i))
      i += 1
    }) invariant {
      0 <= i && i <= size &&
      (i < size ==> buffer.nonFull)
    }

    true
  } else false
}

def insert(b: Buffer): Unit = {
  require(nonFull && b.nonEmpty)
  buffers(nextIndex).set(b)
  nextIndex += 1
}

def encode(b: Buffer): Option[CodeWord] = {
  require(b.nonEmpty)

  var found = false
  var i = 0

  (while (!found && i < nextIndex) {
    if (buffers(i).isEqual(b)) {
      found = true
    } else {
      i += 1
    }
  }) invariant {
    0 <= i && i <= nextIndex && i <= capacity &&
    isValid &&
    (found ==> (i < nextIndex && buffers(i).isEqual(b)))
  }

  if (found) Some(index2CodeWord(i)) else None()
  }
}

@inline // in order to "return" the arrays
def createDictionary() = {
  Dictionary(Array.fill(DictionarySize){ createBuffer() }, 0)
} ensuring { res => res.isEmpty }

def initialise(dict: Dictionary): Unit = {
  require(dict.isEmpty) // initialise only fresh dictionaries

  val buffer = createBuffer()
  assert(buffer.isEmpty)

  var value: Int = Byte.MinValue // Use an Int to avoid overflow issues

  (while (value <= Byte.MaxValue) {
    buffer.append(value.toByte) // no truncation here
    dict.insert(buffer)
    buffer.dropLast()
    value += 1
  }) invariant {
    dict.nonFull &&
    buffer.isEmpty &&
    value >= Byte.MinValue && value <= Byte.MaxValue + 1 // last iteration
  ↪ goes "overflow" on Byte
  }
} ensuring { _ => dict.isValid && dict.nonEmpty }

def encode(fis: FIS, fos: FOS)(implicit state: leon.io.State): Boolean = {
  require(fis.isOpen && fos.isOpen)

  // Initialise the dictionary with the basic alphabet
  val dictionary = createDictionary()
  initialise(dictionary)

  // Small trick to move the static arrays outside the main encoding
  ↪function;
  // this helps analysing the C code in a debugger (less local variables)
  ↪but
  // it actually has no impact on performance (or should, in theory).
  encodeImpl(dictionary, fis, fos)
}

def encodeImpl(dictionary: Dictionary, fis: FIS, fos: FOS)(implicit state:
  ↪ leon.io.State): Boolean = {
  require(fis.isOpen && fos.isOpen && dictionary.nonEmpty)

  var bufferFull = false
  var ioError = false

  val buffer = createBuffer()
```

```
assert(buffer.isEmpty && buffer.nonFull)

var currentOpt = tryReadNext(fis)

// Read from the input file all its content, stop when an error occurs
// (either output error or full buffer)
(while (!bufferFull && !ioError && currentOpt.isDefined) {
  val c = currentOpt.get

  assert(buffer.nonFull)
  buffer.append(c)
  assert(buffer.nonEmpty)

  val code = dictionary.encode(buffer)

  val processBuffer = buffer.isFull || code.isEmpty

  if (processBuffer) {
    // Add s (with c) into the dictionary, if the dictionary size
 ↪limitation allows it
    if (dictionary.nonFull) {
      dictionary.insert(buffer)
    }

    // Encode s (without c) and print it
    buffer.dropLast()
    assert(buffer.nonFull)
    assert(buffer.nonEmpty)
    val code2 = dictionary.encode(buffer)

    assert(code2.isDefined) // (*)
    // To prove (*) we might need to:
    //  - prove the dictionary can encode any 1-length buffer
    //  - the buffer was empty when entering the loop or
    //    that the initial buffer was in the dictionary.
    ioError = !writeCodeWord(fos, code2.get)

    // Prepare for next codeword: set s to c
    buffer.clear()
    buffer.append(c)
    assert(buffer.nonEmpty)
  }

  bufferFull = buffer.isFull

  currentOpt = tryReadNext(fis)
}) invariant {
  bufferFull == buffer.isFull &&
  ((!bufferFull && !ioError) ==> buffer.nonEmpty) // it might always be
 ↪true...
}
```

```
  // Process the remaining buffer
  if (!bufferFull && !ioError) {
    val code = dictionary.encode(buffer)
    assert(code.isDefined) // See (*) above.
    ioError = !writeCodeWord(fos, code.get)
  }

  !bufferFull && !ioError
}

def decode(fis: FIS, fos: FOS)(implicit state: leon.io.State): Boolean = {
  require(fis.isOpen && fos.isOpen)

  // Initialise the dictionary with the basic alphabet
  val dictionary = createDictionary()
  initialise(dictionary)

  decodeImpl(dictionary, fis, fos)
}

def decodeImpl(dictionary: Dictionary, fis: FIS, fos: FOS)(implicit state:
   ↪ leon.io.State): Boolean = {
  require(fis.isOpen && fos.isOpen && dictionary.nonEmpty)

  var illegalInput = false
  var ioError = false
  var bufferFull = false

  var currentOpt = tryReadCodeWord(fis)

  val buffer = createBuffer()

  if (currentOpt.isDefined) {
    val cw = currentOpt.get
    val index = codeWord2Index(cw)

    if (dictionary contains index) {
      bufferFull = !dictionary.appendTo(index, buffer)
      ioError = !writeBytes(fos, buffer)
    } else {
      illegalInput = true
    }

    currentOpt = tryReadCodeWord(fis)
  }

  (while (!illegalInput && !ioError && !bufferFull && currentOpt.isDefined
   ↪) {
    val cw = currentOpt.get
    val index = codeWord2Index(cw)
    val entry = createBuffer()

    if (dictionary contains index) {
      illegalInput = !dictionary.appendTo(index, entry)
    } else if (index == dictionary.lastIndex + 1) {
      entry.set(buffer)
      entry.append(buffer(0))
    } else {
      illegalInput = true
    }

    ioError = !writeBytes(fos, entry)
    bufferFull = buffer.isFull

    if (!bufferFull) {
      val tmp = createBuffer()
      tmp.set(buffer)
      tmp.append(entry(0))
      if (dictionary.nonFull) {
        dictionary.insert(tmp)
      }

      buffer.set(entry)
    }

    currentOpt = tryReadCodeWord(fis)
  }) invariant {
    dictionary.nonEmpty
  }

  !illegalInput && !ioError && !bufferFull
}

sealed abstract class Status
case class Success()     extends Status
case class OpenError()   extends Status
case class EncodeError() extends Status
case class DecodeError() extends Status

implicit def status2boolean(s: Status): Boolean = s match {
  case Success() => true
  case _         => false
}

def _main() = {
  implicit val state = leon.io.newState

  def statusCode(s: Status): Int = s match {
    case Success()     => StdOut.println("success");            0
    case OpenError()   => StdOut.println("couldn't open file"); 1
    case EncodeError() => StdOut.println("encoding failed");    2
    case DecodeError() => StdOut.println("decoding failed");    3
  }
```

```
    def encodeFile(): Status = {
      val input = FIS.open("input.txt")
      val encoded = FOS.open("encoded.txt")

      val res =
        if (input.isOpen && encoded.isOpen) {
          if (encode(input, encoded)) Success()
          else EncodeError()
        } else OpenError()

      encoded.close
      input.close

      res
    }

    def decodeFile(): Status = {
      val encoded = FIS.open("encoded.txt")
      val decoded = FOS.open("decoded.txt")

      val res =
        if (encoded.isOpen && decoded.isOpen) {
          if (decode(encoded, decoded)) Success()
          else DecodeError()
        } else OpenError()

      decoded.close
      encoded.close

      res
    }

    val r1 = encodeFile()
    statusCode(if (r1) decodeFile() else r1)
  }

  @extern
  def main(args: Array[String]): Unit = _main()

}
```

Design B

```
/* Copyright 2009-2016 EPFL, Lausanne */

import leon.lang._
import leon.proof._
import leon.annotation._

import leon.io.{
  FileInputStream => FIS,
  FileOutputStream => FOS,
  StdOut
}

object LZWb {

  // GENERAL NOTES
  // =============
  //
  // Encoding using fixed size of word;
  // Input alphabet is the ASCII range (0-255);
  // A word is made of 16 bits (instead of the classic 12-bit scenario, for
    ↪simplicity);
  // The dictionary is an array of Buffer where the index is the key;

  // We limit the size of the dictionary to an arbitrary size, less than or
    ↪equals to 2^16.
  @inline
  val DictionarySize = 8192 // number of buffers in the dictionary

  @inline
  val DictionaryMemorySize = 524288 // DictionarySize * BufferSize

  // We use fix-sized buffers
  @inline
  val BufferSize = 64 // characters

  val AlphabetSize = Byte.MaxValue + -Byte.MinValue

  private def lemmaSize: Boolean = {
    DictionarySize >= AlphabetSize &&
    BufferSize > 0 &&
    AlphabetSize > 0 &&
    DictionarySize <= 65536 // Cannot encode more index using only 16-bit
      ↪codewords
  }.holds


  // Helper for range equality checking
  private def areRangesEqual(a: Array[Byte], b: Array[Byte], from: Int, to:
    ↪Int): Boolean = {
    require(0 <= from && from <= to && to < a.length && to < b.length)
    a(from) == b(from) && {
      if (from == to) true
      else areRangesEqual(a, b, from + 1, to)
    }
  }

  private def allValidBuffers(buffers: Array[Buffer]): Boolean = {
    def rec(from: Int): Boolean = {
      require(0 <= from && from <= buffers.length)
      if (from < buffers.length) buffers(from).isValid && rec(from + 1)
      else true
    }

    rec(0)
  }

  def allInRange(xs: Array[Int], min: Int, max: Int): Boolean = {
    require(min <= max)

    def rec(index: Int): Boolean = {
      require(0 <= index && index <= xs.length)
      if (xs.length == index) true
      else {
        min <= xs(index) && xs(index) <= max && rec(index + 1)
      }
    }

    rec(0)
  }


  // A buffer representation using a fix-sized array for memory.
  //
  // NOTE Use ‘createBuffer()‘ to get a new buffer; don’t attempt to create
    ↪one yourself.
  case class Buffer(private val array: Array[Byte], private var length: Int)
    ↪ {
    val capacity = array.length
    require(isValid)

    def isValid: Boolean = length >= 0 && length <= capacity && capacity ==
      ↪BufferSize

    def isFull: Boolean = length == capacity
def nonFull: Boolean = length < capacity

def isEmpty: Boolean = length == 0

def nonEmpty: Boolean = length > 0

def isEqual(b: Buffer): Boolean = {
  if (b.length != length) false
  else { isEmpty || areRangesEqual(array, b.array, 0, length - 1) }
}

def isRangeEqual(other: Array[Byte], otherStart: Int, otherSize: Int):
 ↪Boolean = {
  require(0 <= otherStart && 0 <= otherSize && otherSize <= other.length
 ↪ && otherStart <= other.length - otherSize)
  if (size != otherSize) false
  else if (isEmpty) true
  else {
    var i = 0
    var equal = true

    (while (equal && i < size) {
      equal = (other(otherStart + i) == array(i))
      i += 1
    }) invariant (
      0 <= i && i <= size &&
      otherStart + i <= other.length
    )

    equal
  }
}

def size = {
  length
} ensuring { res => 0 <= res && res <= capacity }

def apply(index: Int): Byte = {
  require(index >= 0 && index < length)
  array(index)
}

def append(x: Byte): Unit = {
  require(nonFull)

  array(length) = x

  length += 1
} ensuring { _ => isValid }

def dropLast(): Unit = {
  require(nonEmpty)

    length -= 1
  } ensuring { _ => isValid }

  def clear(): Unit = {
    length = 0
  } ensuring { _ => isEmpty && isValid }

  def set(b: Buffer): Unit = {
    if (b.isEmpty) clear
    else setImpl(b)
  } ensuring { _ => b.isValid && isValid && isEqual(b) }

  private def setImpl(b: Buffer): Unit = {
    require(b.nonEmpty)

    length = b.length

    var i = 0
    (while (i < length) {
      array(i) = b.array(i)
      i += 1
    }) invariant {
      0 <= i && i <= length &&
      // lengthCheckpoint == b.length && lengthCheckpoint == length && //
 ↪no mutation of the length
      isValid && nonEmpty &&
      length == b.length &&
      (i > 0 ==> areRangesEqual(array, b.array, 0, i - 1)) // avoid
 ↪OutOfBoundAccess
    }
  } ensuring { _ => b.isValid && isValid && nonEmpty && isEqual(b) }

}

@inline // very important because we cannot return arrays
def createBuffer(): Buffer = {
  Buffer(Array.fill(BufferSize)(0), 0)
} ensuring { b => b.isEmpty && b.nonFull && b.isValid }


def tryReadNext(fis: FIS)(implicit state: leon.io.State): Option[Byte] = {
  require(fis.isOpen)
  fis.tryReadByte()
}

def writeCodeWord(fos: FOS, cw: CodeWord): Boolean = {
  require(fos.isOpen)
  fos.write(cw.b1) && fos.write(cw.b2)
}
```

```scala
def tryReadCodeWord(fis: FIS)(implicit state: leon.io.State): Option[
   ↪CodeWord] = {
  require(fis.isOpen)
  val b1Opt = fis.tryReadByte()
  val b2Opt = fis.tryReadByte()

  (b1Opt, b2Opt) match {
    case (Some(b1), Some(b2)) => Some(CodeWord(b1, b2))
    case _ => None()
  }
}

def writeBytes(fos: FOS, buffer: Buffer): Boolean = {
  require(fos.isOpen && buffer.nonEmpty)
  var success = true
  var i = 0

  val size = buffer.size

  (while (success && i < size) {
    success = fos.write(buffer(i))
    i += 1
  }) invariant {
    0 <= i && i <= size
  }

  success
}

case class CodeWord(b1: Byte, b2: Byte) // a 16-bit code word

def index2CodeWord(index: Int): CodeWord = {
  require(0 <= index && index < 65536) // unsigned index
  // Shift the index in the range [-32768, 32767] to make it signed
  val signed = index - 32768
  // Split it into two byte components
  val b2 = signed.toByte
  val b1 = (signed >>> 8).toByte
  CodeWord(b1, b2)
}

def codeWord2Index(cw: CodeWord): Int = {
  // When building the signed integer back, make sure to understand
   ↪integer
  // promotion with negative numbers: we need to avoid the signe extension
   ↪ here.
  val signed = (cw.b1 << 8) | (0xff & cw.b2)
  signed + 32768
} ensuring { res => 0 <= res && res < 65536 }
```

```scala
case class Dictionary(private val memory: Array[Byte], private val pteps:
   ↪Array[Int], private var nextIndex: Int) {
  // NOTE ‘pteps‘ stands for Past The End PointerS. It holds the address
   ↪in ‘memory‘ for the next buffer.
  //
  //      By construction, for any index > 0, the begining of the buffer
   ↪is stored in pteps[index - 1].
  //
  //      It therefore holds that the length of the buffer at the given ‘
   ↪index‘ is pteps[index] - pteps[index - 1]
  //      for index > 0, and pteps[0] for index == 0.

  val capacity = pteps.length
  require(
    capacity == DictionarySize &&
    memory.length == DictionaryMemorySize &&
    allInRange(pteps, 0, DictionaryMemorySize) &&
    0 <= nextIndex && nextIndex <= capacity
  )

  def isEmpty = nextIndex == 0

  def nonEmpty = !isEmpty

  def isFull = !nonFull

  def nonFull = {
    nextIndex < capacity && (nextIndex == 0 || (memory.length - pteps(
   ↪nextIndex - 1) >= BufferSize))
  }

  private def getBufferBeginning(index: Int): Int = {
    require(0 <= index && contains(index))
    if (index == 0) 0
    else pteps(index - 1)
  } ensuring { res => 0 <= res && res < DictionaryMemorySize }

  private def getNextBufferBeginning(): Int = {
    require(nonFull) // less equirements than getBufferBeginning
    if (nextIndex == 0) 0
    else pteps(nextIndex - 1)
  } ensuring { res => 0 <= res && res < DictionaryMemorySize }

  private def getBufferSize(index: Int): Int = {
    require(0 <= index && contains(index))
    if (index == 0) pteps(0)
    else pteps(index) - pteps(index - 1)
  } ensuring { res => 0 <= res && res <= BufferSize }

  def lastIndex = {
    require(nonEmpty)
    nextIndex - 1
  } ensuring { res => 0 <= res && res < capacity }

  def contains(index: Int): Boolean = {
    require(0 <= index)
    index < nextIndex
  }

  def appendTo(index: Int, buffer: Buffer): Boolean = {
    require(0 <= index && contains(index))

    val size  = getBufferSize(index)
    val start = getBufferBeginning(index)

    assert(buffer.capacity == BufferSize)
    if (buffer.size < buffer.capacity - size) {
      assert(buffer.nonFull)

      var i = 0
      (while (i < size) {
        buffer.append(memory(start + i))
        i += 1
      }) invariant (
        0 <= i && i <= size &&
        0 <= start && start < DictionaryMemorySize &&
        (i < size ==> buffer.nonFull)
      )

      true
    } else false
  }

  def insert(b: Buffer): Unit = {
    require(nonFull && b.nonEmpty)

    val start = getNextBufferBeginning()

    var i = 0
    (while (i < b.size) {
      memory(start + i) = b(i)
      i += 1
    }) invariant (
      0 <= i && i <= b.size &&
      0 <= start && start < DictionaryMemorySize
    )

    pteps(nextIndex) = start + i

    nextIndex += 1
  }

  def encode(b: Buffer): Option[CodeWord] = {
    require(b.nonEmpty)
```

```scala
    var found = false
    var index = 0

    while (!found && index < nextIndex) {
      val start = getBufferBeginning(index)
      val size  = getBufferSize(index)

      if (b.isRangeEqual(memory, start, size)) {
        found = true
      } else {
        index += 1
      }
    }

    if (found) Some(index2CodeWord(index)) else None()
  }
}

@inline // in order to "return" the arrays
def createDictionary() = {
  Dictionary(Array.fill(DictionaryMemorySize)(0), Array.fill(
   ↪DictionarySize)(0), 0)
} ensuring { res => res.isEmpty }


def initialise(dict: Dictionary): Unit = {
  require(dict.isEmpty) // initialise only fresh dictionaries

  val buffer = createBuffer()
  assert(buffer.isEmpty)

  var value: Int = Byte.MinValue // Use an Int to avoid overflow issues

  (while (value <= Byte.MaxValue) {
    buffer.append(value.toByte) // no truncation here
    dict.insert(buffer)
    buffer.dropLast()
    value += 1
  }) invariant {
    dict.nonFull &&
    buffer.isEmpty &&
    value >= Byte.MinValue && value <= Byte.MaxValue + 1 // last iteration
   ↪ goes "overflow" on Byte
  }
} ensuring { _ => dict.nonEmpty }

def encode(fis: FIS, fos: FOS)(implicit state: leon.io.State): Boolean = {
  require(fis.isOpen && fos.isOpen)

  // Initialise the dictionary with the basic alphabet
  val dictionary = createDictionary()
  initialise(dictionary)

  // Small trick to move the static arrays outside the main encoding
    ↪function;
  // this helps analysing the C code in a debugger (less local variables)
    ↪but
  // it actually has no impact on performance (or should, in theory).
  encodeImpl(dictionary, fis, fos)
}

def encodeImpl(dictionary: Dictionary, fis: FIS, fos: FOS)(implicit state:
    ↪ leon.io.State): Boolean = {
  require(fis.isOpen && fos.isOpen && dictionary.nonEmpty)

  var bufferFull = false
  var ioError = false

  val buffer = createBuffer()
  assert(buffer.isEmpty && buffer.nonFull)

  var currentOpt = tryReadNext(fis)

  // Read from the input file all its content, stop when an error occurs
  // (either output error or full buffer)
  (while (!bufferFull && !ioError && currentOpt.isDefined) {
    val c = currentOpt.get

    assert(buffer.nonFull)
    buffer.append(c)
    assert(buffer.nonEmpty)

    val code = dictionary.encode(buffer)

    val processBuffer = buffer.isFull || code.isEmpty

    if (processBuffer) {
      // Add s (with c) into the dictionary, if the dictionary size
    ↪limitation allows it
      if (dictionary.nonFull) {
        dictionary.insert(buffer)
      }

      // Encode s (without c) and print it
      buffer.dropLast()
      assert(buffer.nonFull)
      assert(buffer.nonEmpty)
      val code2 = dictionary.encode(buffer)

      assert(code2.isDefined) // (*)
      // To prove (*) we might need to:
      //  - prove the dictionary can encode any 1-length buffer
      //  - the buffer was empty when entering the loop or
      //    that the initial buffer was in the dictionary.
      ioError = !writeCodeWord(fos, code2.get)

      // Prepare for next codeword: set s to c
      buffer.clear()
      buffer.append(c)
      assert(buffer.nonEmpty)
    }

    bufferFull = buffer.isFull

    currentOpt = tryReadNext(fis)
  }) invariant {
    bufferFull == buffer.isFull &&
    ((!bufferFull && !ioError) ==> buffer.nonEmpty) // it might always be
  ↪true...
  }

  // Process the remaining buffer
  if (!bufferFull && !ioError) {
    val code = dictionary.encode(buffer)
    assert(code.isDefined) // See (*) above.
    ioError = !writeCodeWord(fos, code.get)
  }

  !bufferFull && !ioError
}

def decode(fis: FIS, fos: FOS)(implicit state: leon.io.State): Boolean = {
  require(fis.isOpen && fos.isOpen)

  // Initialise the dictionary with the basic alphabet
  val dictionary = createDictionary()
  initialise(dictionary)

  decodeImpl(dictionary, fis, fos)
}

def decodeImpl(dictionary: Dictionary, fis: FIS, fos: FOS)(implicit state:
    ↪ leon.io.State): Boolean = {
  require(fis.isOpen && fos.isOpen && dictionary.nonEmpty)

  var illegalInput = false
  var ioError = false
  var bufferFull = false

  var currentOpt = tryReadCodeWord(fis)

  val buffer = createBuffer()

  if (currentOpt.isDefined) {
    val cw = currentOpt.get
    val index = codeWord2Index(cw)

    if (dictionary contains index) {
      bufferFull = !dictionary.appendTo(index, buffer)
      ioError = !writeBytes(fos, buffer)
    } else {
      illegalInput = true
    }

    currentOpt = tryReadCodeWord(fis)
  }

  (while (!illegalInput && !ioError && !bufferFull && currentOpt.isDefined
   ↪) {
    val cw = currentOpt.get
    val index = codeWord2Index(cw)
    val entry = createBuffer()

    if (dictionary contains index) {
      illegalInput = !dictionary.appendTo(index, entry)
    } else if (index == dictionary.lastIndex + 1) {
      entry.set(buffer)
      entry.append(buffer(0))
    } else {
      illegalInput = true
    }

    ioError = !writeBytes(fos, entry)
    bufferFull = buffer.isFull

    if (!bufferFull) {
      val tmp = createBuffer()
      tmp.set(buffer)
      tmp.append(entry(0))
      if (dictionary.nonFull) {
        dictionary.insert(tmp)
      }

      buffer.set(entry)
    }

    currentOpt = tryReadCodeWord(fis)
  }) invariant {
    dictionary.nonEmpty
  }

  !illegalInput && !ioError && !bufferFull
}


sealed abstract class Status
case class Success()     extends Status
```

```
case class OpenError()   extends Status
case class EncodeError() extends Status
case class DecodeError() extends Status

implicit def status2boolean(s: Status): Boolean = s match {
  case Success() => true
  case _         => false
}

def _main() = {
  implicit val state = leon.io.newState

  def statusCode(s: Status): Int = s match {
    case Success()     => StdOut.println("success");            0
    case OpenError()   => StdOut.println("couldn't open file"); 1
    case EncodeError() => StdOut.println("encoding failed");    2
    case DecodeError() => StdOut.println("decoding failed");    3
  }

  def encodeFile(): Status = {
    val input = FIS.open("input.txt")
    val encoded = FOS.open("encoded.txt")

    val res =
      if (input.isOpen && encoded.isOpen) {
        if (encode(input, encoded)) Success()
        else EncodeError()
      } else OpenError()

    encoded.close
    input.close

    res
  }

  def decodeFile(): Status = {
    val encoded = FIS.open("encoded.txt")
    val decoded = FOS.open("decoded.txt")

    val res =
      if (encoded.isOpen && decoded.isOpen) {
        if (decode(encoded, decoded)) Success()
        else DecodeError()
      } else OpenError()

    decoded.close
    encoded.close

    res
  }

  val r1 = encodeFile()
  statusCode(if (r1) decodeFile() else r1)
}

@extern
```

```
  def main(args: Array[String]): Unit = _main()

}
```

B.2 Generated C99 Code

Design A

The generated code is unfortunately too long to decently fit in this annex for reasons detailed in Section 9.3. It can however be accessed here: https://github.com/mantognini/GenC.

Design B

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

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

/* -------------------------------- type aliases ----- */

typedef FILE* FileInputStream;
typedef void* State;
typedef FILE* FileOutputStream;

/* --------------------------------------- enums ----- */

typedef enum {
  tag_None_0_CodeWord_0,
  tag_Some_0_CodeWord_0
} enum_Option_0_CodeWord_0;

typedef enum {
  tag_None_0_int8,
  tag_Some_0_int8
} enum_Option_0_int8;

typedef enum {
  tag_OpenError_0,
  tag_DecodeError_0,
  tag_Success_0,
  tag_EncodeError_0
} enum_Status_0;

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

typedef struct {
  int8_t* data;
  int32_t length;
} array_int8;

typedef struct {
  int32_t* data;
  int32_t length;
} array_int32;

typedef struct {
  array_int8 memory_1;
  array_int32 pteps_1;
  int32_t nextIndex_1;
} Dictionary_2;

typedef struct {
  array_int8 array_1;
  int32_t length_3;
} Buffer_2;

typedef struct {
  int8_t extra;
} None_0_CodeWord_0;

typedef struct {
  int8_t b1_0;
  int8_t b2_0;
} CodeWord_0;

typedef struct {
  CodeWord_0 v_13;
} Some_0_CodeWord_0;

typedef union {
  None_0_CodeWord_0 None_0_CodeWord_0_v;
  Some_0_CodeWord_0 Some_0_CodeWord_0_v;
} union_Option_0_CodeWord_0;

typedef struct {
  enum_Option_0_CodeWord_0 tag;
  union_Option_0_CodeWord_0 value;
} Option_0_CodeWord_0;

typedef struct {
  int8_t extra;
} None_0_int8;

typedef struct {
  int8_t v_13;
} Some_0_int8;

typedef union {
  None_0_int8 None_0_int8_v;
  Some_0_int8 Some_0_int8_v;
} union_Option_0_int8;

typedef struct {
  enum_Option_0_int8 tag;
  union_Option_0_int8 value;
} Option_0_int8;

typedef struct {
  Option_0_int8 _1;
  Option_0_int8 _2;
} Tuple_Option_0_int8_Option_0_int8;

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

static bool writeBytes_2(FileOutputStream fos_6, Buffer_2* buffer_7);
static void setImpl_3(Buffer_2* thiss_821, Buffer_2* b_31);
static Option_0_int8 tryReadByte_3(FileInputStream thiss_132, State state_27
      ↪);
static bool isEmpty_12(Buffer_2* thiss_787);
static bool appendTo_3(Dictionary_2* thiss_819, int32_t index_33, Buffer_2*
      ↪buffer_11);
static bool isEmpty_8_CodeWord_0(Option_0_CodeWord_0 thiss_109);
static void initialise_2(Dictionary_2* dict_2);
static enum_Status_0 decodeFile_1(State* state_24);
static int32_t size_10(Buffer_2* thiss_767);
static bool isOpen_4(FileInputStream thiss_130);
static bool encode_4(FileInputStream fis_2, FileOutputStream fos_2, State
      ↪state_7);
static void print_5(char c_3);
static int32_t capacity_7(Dictionary_2* thiss_755);
static FileOutputStream open_2(char* filename_2);
static void println_6(void);
static void clear_4(Buffer_2* thiss_799);
static Option_0_CodeWord_0 tryReadCodeWord_1(FileInputStream fis_1, State
      ↪state_6);
static void set_8(Buffer_2* thiss_817, Buffer_2* b_29);
static int8_t get_5_int8(Option_0_int8 thiss_106);
static bool contains_5(Dictionary_2* thiss_800, int32_t index_31);
static bool isRangeEqual_3(Buffer_2* thiss_797, array_int8 other_2, int32_t
      ↪otherStart_2, int32_t otherSize_2);
static CodeWord_0 get_5_CodeWord_0(Option_0_CodeWord_0 thiss_106);
static Option_0_int8 tryReadNext_1(FileInputStream fis_0, State state_5);
static int32_t _main(void);
static bool isEmpty_8_int8(Option_0_int8 thiss_109);
static bool encodeImpl_2(Dictionary_2* dictionary_5, FileInputStream fis_6,
      ↪FileOutputStream fos_7, State state_30);
static int32_t lastIndex_3(Dictionary_2* thiss_818);
static int32_t capacity_5(Buffer_2* thiss_753);
static bool nonFull_6(Dictionary_2* thiss_780);
static void append_3(Buffer_2* thiss_781, int8_t x_289);
static bool decodeImpl_2(Dictionary_2* dictionary_8, FileInputStream fis_7,
      ↪FileOutputStream fos_8, State state_31);
static bool write_8(FileOutputStream thiss_125, int8_t x_246);
 int32_t main(int32_t argc, char** argv);
static FileInputStream open_3(char* filename_3, State state_23);
static int8_t apply_21(Buffer_2* thiss_769, int32_t index_29);
static int32_t statusCode_1(State* state_24, enum_Status_0 s_20);
static bool isFull_5(Buffer_2* thiss_796);
static void insert_3(Dictionary_2* thiss_786, Buffer_2* b_23);
static bool isDefined_2_CodeWord_0(Option_0_CodeWord_0 thiss_111);
static Option_0_CodeWord_0 encode_5(Dictionary_2* thiss_798, Buffer_2* b_25)
      ↪;
static int32_t getBufferBeginning_3(Dictionary_2* thiss_801, int32_t
      ↪index_32);
static bool writeCodeWord_1(FileOutputStream fos_0, CodeWord_0 cw_0);
static bool close_5(FileOutputStream thiss_123);
static int32_t codeWord2Index_1(CodeWord_0 cw_1);
static bool decode_1(FileInputStream fis_4, FileOutputStream fos_4, State
      ↪state_9);
static bool isDefined_2_int8(Option_0_int8 thiss_111);
static CodeWord_0 index2CodeWord_1(int32_t index_3);
static bool close_4(FileInputStream thiss_129, State state_25);
static void println_5(char* s_19);
static int32_t getNextBufferBeginning_3(Dictionary_2* thiss_783);
static bool isOpen_5(FileOutputStream thiss_124);
static int8_t impl_4(FileInputStream* thiss_132, State* state_27, bool*
      ↪valid_2);
static void print_4(char* x_17);
static void dropLast_3(Buffer_2* thiss_785);
static enum_Status_0 encodeFile_1(State* state_24);
static int32_t getBufferSize_3(Dictionary_2* thiss_795, int32_t index_30);
static State newState_2(void);

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

static bool writeBytes_2(FileOutputStream fos_6, Buffer_2* buffer_7) {
    bool success_0 = true;
    int32_t i_20 = 0;
    int32_t size_3 = size_10(buffer_7);
    while (success_0 && i_20 < size_3) {
        FileOutputStream norm_92 = fos_6;
        int8_t norm_91 = apply_21(buffer_7, i_20);
        int8_t norm_93 = norm_91;
        bool norm_94 = write_8(norm_92, norm_93);
        success_0 = norm_94;
        i_20 = i_20 + 1;
    }
    return success_0;
}

static void setImpl_3(Buffer_2* thiss_821, Buffer_2* b_31) {
    thiss_821->length_3 = b_31->length_3;
    int32_t i_19 = 0;
    while (i_19 < thiss_821->length_3) {
        thiss_821->array_1.data[i_19] = b_31->array_1.data[i_19];
        i_19 = i_19 + 1;
    }
}

static Option_0_int8 tryReadByte_3(FileInputStream thiss_132, State state_27
     ↪) {
    bool valid_2 = true;
    int8_t res_18 = impl_4(&thiss_132, &state_27, &valid_2);
    if (valid_2) {
        return (Option_0_int8) { .tag = tag_Some_0_int8, .value = (
     ↪union_Option_0_int8) { .Some_0_int8_v = (Some_0_int8) { .v_13 =
     ↪res_18 } } };
    } else {
        return (Option_0_int8) { .tag = tag_None_0_int8, .value = (
     ↪union_Option_0_int8) { .None_0_int8_v = (None_0_int8) { .extra = 0 }
     ↪} };
    }
}

static bool isEmpty_12(Buffer_2* thiss_787) {
    return thiss_787->length_3 == 0;
}

static bool appendTo_3(Dictionary_2* thiss_819, int32_t index_33, Buffer_2*
     ↪buffer_11) {
    int32_t size_4 = getBufferSize_3(thiss_819, index_33);
    int32_t start_1 = getBufferBeginning_3(thiss_819, index_33);
    int32_t norm_84 = size_10(buffer_11);
    int32_t norm_88 = norm_84;
    int32_t norm_85 = capacity_5(buffer_11);
    int32_t norm_86 = norm_85;
    int32_t norm_87 = size_4;
    int32_t norm_89 = norm_86 - norm_87;
    if (norm_88 < norm_89) {
        int32_t i_21 = 0;
        while (i_21 < size_4) {
            append_3(buffer_11, thiss_819->memory_1.data[start_1 + i_21]);
            i_21 = i_21 + 1;
        }
        return true;
    } else {
        return false;
    }
}

static bool isEmpty_8_CodeWord_0(Option_0_CodeWord_0 thiss_109) {
    if (thiss_109.tag == tag_Some_0_CodeWord_0) {
        return false;
    } else if (thiss_109.tag == tag_None_0_CodeWord_0) {
        return true;
    }
}

static void initialise_2(Dictionary_2* dict_2) {
    int8_t leon_buffer_7[64] = { 0 };
    array_int8 norm_10 = (array_int8) { .data = leon_buffer_7, .length = 64
     ↪};
    array_int8* norm_11 = &norm_10;
    int32_t norm_12 = 0;
    Buffer_2 res_217 = (Buffer_2) { .array_1 = (*norm_11), .length_3 =
     ↪norm_12 };
    Buffer_2* buffer_8 = &res_217;
    int32_t value_2 = -128;
    while (value_2 <= ((int32_t)127)) {
        append_3(buffer_8, (int8_t)value_2);
        insert_3(dict_2, buffer_8);
        dropLast_3(buffer_8);
        value_2 = value_2 + 1;
    }
}

static enum_Status_0 decodeFile_1(State* state_24) {
    FileInputStream encoded_0 = open_3("encoded.txt", *state_24);
    FileOutputStream decoded_0 = open_2("decoded.txt");
    bool norm_70 = isOpen_4(encoded_0);
    bool norm_72 = norm_70;
    bool norm_74;
    if (norm_72) {
        bool norm_71 = isOpen_5(decoded_0);
        bool norm_73 = norm_71;
        norm_74 = norm_73;
    } else {
        norm_74 = false;
    }
    enum_Status_0 norm_128;
    if (norm_74) {
        if (decode_1(encoded_0, decoded_0, *state_24)) {
            norm_128 = tag_Success_0;
        } else {
            norm_128 = tag_DecodeError_0;
        }
    } else {
        norm_128 = tag_OpenError_0;
    }
    enum_Status_0 res_8 = norm_128;
    close_5(decoded_0);
    close_4(encoded_0, *state_24);
    return res_8;
}

static int32_t size_10(Buffer_2* thiss_767) {
    return thiss_767->length_3;
}


static bool isOpen_4(FILE* this) {
  return this != NULL;
}


static bool encode_4(FileInputStream fis_2, FileOutputStream fos_2, State
     ↪state_7) {
    int8_t leon_buffer_5[524288] = { 0 };
    array_int8 norm_5 = (array_int8) { .data = leon_buffer_5, .length =
     ↪524288 };
    array_int8* norm_7 = &norm_5;
    int32_t leon_buffer_6[8192] = { 0 };
    array_int32 norm_6 = (array_int32) { .data = leon_buffer_6, .length =
     ↪8192 };
    array_int32* norm_8 = &norm_6;
    int32_t norm_9 = 0;
    Dictionary_2 res_219 = (Dictionary_2) { .memory_1 = (*norm_7), .pteps_1
     ↪= (*norm_8), .nextIndex_1 = norm_9 };
    Dictionary_2* dictionary_6 = &res_219;
    initialise_2(dictionary_6);
    return encodeImpl_2(dictionary_6, fis_2, fos_2, state_7);
}


static void print_5(char c) {
  printf("%c", c);
}


static int32_t capacity_7(Dictionary_2* thiss_755) {
    return thiss_755->pteps_1.length;
}


static FILE* open_2(char* filename) {
  FILE* this = fopen(filename, "w");
  /* this == NULL on failure */
  return this;
}


static void println_6(void) {
    print_5('\n');
}

static void clear_4(Buffer_2* thiss_799) {
    thiss_799->length_3 = 0;
}

static Option_0_CodeWord_0 tryReadCodeWord_1(FileInputStream fis_1, State
     ↪state_6) {
    Option_0_int8 b1Opt_0 = tryReadByte_3(fis_1, state_6);
    Option_0_int8 b2Opt_0 = tryReadByte_3(fis_1, state_6);
    Tuple_Option_0_int8_Option_0_int8 tmp_2 = (
     ↪Tuple_Option_0_int8_Option_0_int8) { ._1 = b1Opt_0, ._2 = b2Opt_0 };
    if (tmp_2._1.tag == tag_Some_0_int8 && tmp_2._2.tag == tag_Some_0_int8)
     ↪{
        return (Option_0_CodeWord_0) { .tag = tag_Some_0_CodeWord_0, .value
     ↪= (union_Option_0_CodeWord_0) { .Some_0_CodeWord_0_v = (
     ↪Some_0_CodeWord_0) { .v_13 = (CodeWord_0) { .b1_0 = tmp_2._1.value.
     ↪Some_0_int8_v.v_13, .b2_0 = tmp_2._2.value.Some_0_int8_v.v_13 } } }
     ↪};
    } else {
        return (Option_0_CodeWord_0) { .tag = tag_None_0_CodeWord_0, .value
     ↪= (union_Option_0_CodeWord_0) { .None_0_CodeWord_0_v = (
     ↪None_0_CodeWord_0) { .extra = 0 } } };
    }
}

static void set_8(Buffer_2* thiss_817, Buffer_2* b_29) {
    if (isEmpty_12(b_29)) {
        clear_4(thiss_817);
    } else {
        setImpl_3(thiss_817, b_29);
    }
}

static int8_t get_5_int8(Option_0_int8 thiss_106) {
    return thiss_106.value.Some_0_int8_v.v_13;
}

static bool contains_5(Dictionary_2* thiss_800, int32_t index_31) {
    return index_31 < thiss_800->nextIndex_1;
}

static bool isRangeEqual_3(Buffer_2* thiss_797, array_int8 other_2, int32_t
     ↪otherStart_2, int32_t otherSize_2) {
    int32_t norm_30 = size_10(thiss_797);
```

```c
    int32_t norm_31 = norm_30;
    int32_t norm_32 = otherSize_2;
    if (norm_31 != norm_32) {
        return false;
    } else if (isEmpty_12(thiss_797)) {
        return true;
    } else {
        int32_t i_18 = 0;
        bool equal_0 = true;
        while (true) {
            bool norm_36 = equal_0;
            bool norm_38;
            if (norm_36) {
                int32_t norm_34 = i_18;
                int32_t norm_33 = size_10(thiss_797);
                int32_t norm_35 = norm_33;
                bool norm_37 = norm_34 < norm_35;
                norm_38 = norm_37;
            } else {
                norm_38 = false;
            }
            if (norm_38) {
                equal_0 = ((int32_t)other_2.data[otherStart_2 + i_18]) == ((
  ↪int32_t)thiss_797->array_1.data[i_18]);
                i_18 = i_18 + 1;
            } else {
                break;
            }
        }
        return equal_0;
    }
}

static CodeWord_0 get_5_CodeWord_0(Option_0_CodeWord_0 thiss_106) {
    return thiss_106.value.Some_0_CodeWord_0_v.v_13;
}

static Option_0_int8 tryReadNext_1(FileInputStream fis_0, State state_5) {
    return tryReadByte_3(fis_0, state_5);
}

static int32_t _main(void) {
    State state_24 = newState_2();
    enum_Status_0 r1_0 = encodeFile_1(&state_24);
    bool norm_69;
    if (r1_0 == tag_Success_0) {
        norm_69 = true;
    } else {
        norm_69 = false;
    }
    enum_Status_0 norm_129;
    if (norm_69) {
        norm_129 = decodeFile_1(&state_24);
    } else {
        norm_129 = r1_0;
    }
    enum_Status_0 norm_130 = norm_129;
    return statusCode_1(&state_24, norm_130);
}

static bool isEmpty_8_int8(Option_0_int8 thiss_109) {
    if (thiss_109.tag == tag_Some_0_int8) {
        return false;
    } else if (thiss_109.tag == tag_None_0_int8) {
        return true;
    }
}

static bool encodeImpl_2(Dictionary_2* dictionary_5, FileInputStream fis_6,
    ↪FileOutputStream fos_7, State state_30) {
    bool bufferFull_0 = false;
    bool ioError_0 = false;
    int8_t leon_buffer_8[64] = { 0 };
    array_int8 norm_19 = (array_int8) { .data = leon_buffer_8, .length = 64
    ↪};
    array_int8* norm_20 = &norm_19;
    int32_t norm_21 = 0;
    Buffer_2 res_218 = (Buffer_2) { .array_1 = (*norm_20), .length_3 =
    ↪norm_21 };
    Buffer_2* buffer_9 = &res_218;
    Option_0_int8 currentOpt_0 = tryReadNext_1(fis_6, state_30);
    while (true) {
        bool norm_27 = !bufferFull_0;
        bool norm_29;
        if (norm_27) {
            bool norm_24 = !ioError_0;
            bool norm_26;
            if (norm_24) {
                bool norm_23 = isDefined_2_int8(currentOpt_0);
                bool norm_25 = norm_23;
                norm_26 = norm_25;
            } else {
                norm_26 = false;
            }
            bool norm_28 = norm_26;
            norm_29 = norm_28;
        } else {
            norm_29 = false;
        }
        if (norm_29) {
            int8_t c_5 = get_5_int8(currentOpt_0);
            append_3(buffer_9, c_5);
            Option_0_CodeWord_0 code_1 = encode_5(dictionary_5, buffer_9);
            bool norm_44 = isFull_5(buffer_9);
            bool norm_46 = norm_44;
            bool norm_48;
            if (norm_46) {
                norm_48 = true;
            } else {
                bool norm_45 = isEmpty_8_CodeWord_0(code_1);
                bool norm_47 = norm_45;
                norm_48 = norm_47;
            }
            bool processBuffer_0 = norm_48;
            if (processBuffer_0) {
                if (nonFull_6(dictionary_5)) {
                    insert_3(dictionary_5, buffer_9);
                } else {

                }
                dropLast_3(buffer_9);
                Option_0_CodeWord_0 code2_0 = encode_5(dictionary_5,
      ↪buffer_9);
                FileOutputStream norm_59 = fos_7;
                CodeWord_0 norm_58 = get_5_CodeWord_0(code2_0);
                CodeWord_0 norm_60 = norm_58;
                bool norm_61 = writeCodeWord_1(norm_59, norm_60);
                ioError_0 = !norm_61;
                clear_4(buffer_9);
                append_3(buffer_9, c_5);
            } else {

            }
            bool norm_62 = isFull_5(buffer_9);
            bufferFull_0 = norm_62;
            Option_0_int8 norm_63 = tryReadNext_1(fis_6, state_30);
            currentOpt_0 = norm_63;
        } else {
            break;
        }
    }
    if (!bufferFull_0 && !ioError_0) {
        Option_0_CodeWord_0 code_2 = encode_5(dictionary_5, buffer_9);
        FileOutputStream norm_65 = fos_7;
        CodeWord_0 norm_64 = get_5_CodeWord_0(code_2);
        CodeWord_0 norm_66 = norm_64;
        bool norm_67 = writeCodeWord_1(norm_65, norm_66);
        ioError_0 = !norm_67;
    } else {

    }
    return !bufferFull_0 && !ioError_0;
}

static int32_t lastIndex_3(Dictionary_2* thiss_818) {
    return thiss_818->nextIndex_1 - 1;
```

```
}

static int32_t capacity_5(Buffer_2* thiss_753) {
    return thiss_753->array_1.length;
}

static bool nonFull_6(Dictionary_2* thiss_780) {
    int32_t norm_50 = thiss_780->nextIndex_1;
    int32_t norm_49 = capacity_7(thiss_780);
    int32_t norm_51 = norm_49;
    bool norm_52 = norm_50 < norm_51;
    if (norm_52) {
        bool norm_53 = thiss_780->nextIndex_1 == 0 || thiss_780->memory_1.
      ↪length - thiss_780->pteps_1.data[thiss_780->nextIndex_1 - 1] >= 64;
        return norm_53;
    } else {
        return false;
    }
}

static void append_3(Buffer_2* thiss_781, int8_t x_289) {
    thiss_781->array_1.data[thiss_781->length_3] = x_289;
    thiss_781->length_3 = thiss_781->length_3 + 1;
}

static bool decodeImpl_2(Dictionary_2* dictionary_8, FileInputStream fis_7,
      ↪FileOutputStream fos_8, State state_31) {
    bool illegalInput_0 = false;
    bool ioError_1 = false;
    bool bufferFull_1 = false;
    Option_0_CodeWord_0 currentOpt_1 = tryReadCodeWord_1(fis_7, state_31);
    int8_t leon_buffer_0[64] = { 0 };
    array_int8 norm_80 = (array_int8) { .data = leon_buffer_0, .length = 64
      ↪};
    array_int8* norm_81 = &norm_80;
    int32_t norm_82 = 0;
    Buffer_2 res_220 = (Buffer_2) { .array_1 = (*norm_81), .length_3 =
      ↪norm_82 };
    Buffer_2* buffer_12 = &res_220;
    if (isDefined_2_CodeWord_0(currentOpt_1)) {
        CodeWord_0 cw_2 = get_5_CodeWord_0(currentOpt_1);
        int32_t index_10 = codeWord2Index_1(cw_2);
        if (contains_5(dictionary_8, index_10)) {
            bool norm_90 = appendTo_3(dictionary_8, index_10, buffer_12);
            bufferFull_1 = !norm_90;
            bool norm_95 = writeBytes_2(fos_8, buffer_12);
            ioError_1 = !norm_95;
        } else {
            illegalInput_0 = true;
        }
        Option_0_CodeWord_0 norm_96 = tryReadCodeWord_1(fis_7, state_31);
        currentOpt_1 = norm_96;
    } else {

    }
    while (true) {
        bool norm_104 = !illegalInput_0;
        bool norm_106;
        if (norm_104) {
            bool norm_101 = !ioError_1;
            bool norm_103;
            if (norm_101) {
                bool norm_98 = !bufferFull_1;
                bool norm_100;
                if (norm_98) {
                    bool norm_97 = isDefined_2_CodeWord_0(currentOpt_1);
                    bool norm_99 = norm_97;
                    norm_100 = norm_99;
                } else {
                    norm_100 = false;
                }
                bool norm_102 = norm_100;
                norm_103 = norm_102;
            } else {
                norm_103 = false;
            }
            bool norm_105 = norm_103;
            norm_106 = norm_105;
        } else {
            norm_106 = false;
        }
        if (norm_106) {
            CodeWord_0 cw_3 = get_5_CodeWord_0(currentOpt_1);
            int32_t index_11 = codeWord2Index_1(cw_3);
            int8_t leon_buffer_1[64] = { 0 };
            array_int8 norm_107 = (array_int8) { .data = leon_buffer_1, .
↪length = 64 };
            array_int8* norm_108 = &norm_107;
            int32_t norm_109 = 0;
            Buffer_2 res_221 = (Buffer_2) { .array_1 = (*norm_108), .
↪length_3 = norm_109 };
            Buffer_2* entry_1 = &res_221;
            if (contains_5(dictionary_8, index_11)) {
                bool norm_110 = appendTo_3(dictionary_8, index_11, entry_1);
                illegalInput_0 = !norm_110;
            } else {
                int32_t norm_114 = index_11;
                int32_t norm_111 = lastIndex_3(dictionary_8);
                int32_t norm_112 = norm_111;
                int32_t norm_113 = 1;
                int32_t norm_115 = norm_112 + norm_113;
                if (norm_114 == norm_115) {
                    set_8(entry_1, buffer_12);
                    Buffer_2* norm_117 = entry_1;
                    int8_t norm_116 = apply_21(buffer_12, 0);
                    int8_t norm_118 = norm_116;
                    append_3(norm_117, norm_118);
                } else {
                    illegalInput_0 = true;
                }
            }
            bool norm_119 = writeBytes_2(fos_8, entry_1);
            ioError_1 = !norm_119;
            bool norm_120 = isFull_5(buffer_12);
            bufferFull_1 = norm_120;
            if (!bufferFull_1) {
                int8_t leon_buffer_2[64] = { 0 };
                array_int8 norm_121 = (array_int8) { .data = leon_buffer_2,
↪.length = 64 };
                array_int8* norm_122 = &norm_121;
                int32_t norm_123 = 0;
                Buffer_2 res_222 = (Buffer_2) { .array_1 = (*norm_122), .
↪length_3 = norm_123 };
                Buffer_2* tmp_1 = &res_222;
                set_8(tmp_1, buffer_12);
                Buffer_2* norm_125 = tmp_1;
                int8_t norm_124 = apply_21(entry_1, 0);
                int8_t norm_126 = norm_124;
                append_3(norm_125, norm_126);
                if (nonFull_6(dictionary_8)) {
                    insert_3(dictionary_8, tmp_1);
                } else {

                }
                set_8(buffer_12, entry_1);
            } else {

            }
            Option_0_CodeWord_0 norm_127 = tryReadCodeWord_1(fis_7, state_31
↪);
            currentOpt_1 = norm_127;
        } else {
            break;
        }
    }
    return !illegalInput_0 && (!ioError_1 && !bufferFull_1);
}


static bool write_8(FILE* this, int8_t x) {
  return fprintf(this, "%c", x) >= 0;
}


 int32_t main(int32_t argc, char** argv) {
    return _main();
```

```
}


static FILE* open_3(char* filename, void* unused) {
  FILE* this = fopen(filename, "r");
  /* this == NULL on failure */
  return this;
}


static int8_t apply_21(Buffer_2* thiss_769, int32_t index_29) {
    return thiss_769->array_1.data[index_29];
}

static int32_t statusCode_1(State* state_24, enum_Status_0 s_20) {
    if (s_20 == tag_Success_0) {
        println_5("success");
        return 0;
    } else if (s_20 == tag_OpenError_0) {
        println_5("couldn't_open_file");
        return 1;
    } else if (s_20 == tag_EncodeError_0) {
        println_5("encoding_failed");
        return 2;
    } else if (s_20 == tag_DecodeError_0) {
        println_5("decoding_failed");
        return 3;
    }
}

static bool isFull_5(Buffer_2* thiss_796) {
    int32_t norm_42 = thiss_796->length_3;
    int32_t norm_41 = capacity_5(thiss_796);
    int32_t norm_43 = norm_41;
    return norm_42 == norm_43;
}

static void insert_3(Dictionary_2* thiss_786, Buffer_2* b_23) {
    int32_t start_2 = getNextBufferBeginning_3(thiss_786);
    int32_t i_22 = 0;
    while (true) {
        int32_t norm_14 = i_22;
        int32_t norm_13 = size_10(b_23);
        int32_t norm_15 = norm_13;
        if (norm_14 < norm_15) {
            int32_t norm_17 = start_2 + i_22;
            int8_t norm_16 = apply_21(b_23, i_22);
            int8_t norm_18 = norm_16;
            thiss_786->memory_1.data[norm_17] = norm_18;
            i_22 = i_22 + 1;
        } else {
            break;
        }
    }
    thiss_786->pteps_1.data[thiss_786->nextIndex_1] = start_2 + i_22;
    thiss_786->nextIndex_1 = thiss_786->nextIndex_1 + 1;
}

static bool isDefined_2_CodeWord_0(Option_0_CodeWord_0 thiss_111) {
    bool norm_83 = isEmpty_8_CodeWord_0(thiss_111);
    return !norm_83;
}

static Option_0_CodeWord_0 encode_5(Dictionary_2* thiss_798, Buffer_2* b_25)
    ↪ {
    bool found_0 = false;
    int32_t index_9 = 0;
    while (!found_0 && index_9 < thiss_798->nextIndex_1) {
        int32_t start_3 = getBufferBeginning_3(thiss_798, index_9);
        int32_t size_5 = getBufferSize_3(thiss_798, index_9);
        if (isRangeEqual_3(b_25, thiss_798->memory_1, start_3, size_5)) {
            found_0 = true;
        } else {
            index_9 = index_9 + 1;
        }
    }
    if (found_0) {
        CodeWord_0 norm_39 = index2CodeWord_1(index_9);
        CodeWord_0 norm_40 = norm_39;
        return (Option_0_CodeWord_0) { .tag = tag_Some_0_CodeWord_0, .value
    ↪= (union_Option_0_CodeWord_0) { .Some_0_CodeWord_0_v = (
    ↪Some_0_CodeWord_0) { .v_13 = norm_40 } } };
    } else {
        return (Option_0_CodeWord_0) { .tag = tag_None_0_CodeWord_0, .value
    ↪= (union_Option_0_CodeWord_0) { .None_0_CodeWord_0_v = (
    ↪None_0_CodeWord_0) { .extra = 0 } } };
    }
}

static int32_t getBufferBeginning_3(Dictionary_2* thiss_801, int32_t
    ↪index_32) {
    if (index_32 == 0) {
        return 0;
    } else {
        return thiss_801->pteps_1.data[index_32 - 1];
    }
}

static bool writeCodeWord_1(FileOutputStream fos_0, CodeWord_0 cw_0) {
    bool norm_54 = write_8(fos_0, cw_0.b1_0);
    bool norm_56 = norm_54;
    if (norm_56) {
        bool norm_55 = write_8(fos_0, cw_0.b2_0);
        bool norm_57 = norm_55;
        return norm_57;
    } else {
        return false;
    }
}


static bool close_5(FILE* this) {
  if (this != NULL)
    return fclose(this) == 0;
  else
    return true;
}


static int32_t codeWord2Index_1(CodeWord_0 cw_1) {
    int32_t signed_1 = ((int32_t)(((uint32_t)((int32_t)cw_1.b1_0)) << 8)) |
     ↪(255 & ((int32_t)cw_1.b2_0));
    return signed_1 + 32768;
}

static bool decode_1(FileInputStream fis_4, FileOutputStream fos_4, State
     ↪state_9) {
    int8_t leon_buffer_3[524288] = { 0 };
    array_int8 norm_75 = (array_int8) { .data = leon_buffer_3, .length =
     ↪524288 };
    array_int8* norm_77 = &norm_75;
    int32_t leon_buffer_4[8192] = { 0 };
    array_int32 norm_76 = (array_int32) { .data = leon_buffer_4, .length =
     ↪8192 };
    array_int32* norm_78 = &norm_76;
    int32_t norm_79 = 0;
    Dictionary_2 res_223 = (Dictionary_2) { .memory_1 = (*norm_77), .pteps_1
     ↪ = (*norm_78), .nextIndex_1 = norm_79 };
    Dictionary_2* dictionary_9 = &res_223;
    initialise_2(dictionary_9);
    return decodeImpl_2(dictionary_9, fis_4, fos_4, state_9);
}

static bool isDefined_2_int8(Option_0_int8 thiss_111) {
    bool norm_22 = isEmpty_8_int8(thiss_111);
    return !norm_22;
}

static CodeWord_0 index2CodeWord_1(int32_t index_3) {
    int32_t signed_0 = index_3 - 32768;
    int8_t b2_2 = (int8_t)signed_0;
    int8_t b1_2 = (int8_t)((int32_t)(((uint32_t)signed_0) >> 8));
    return (CodeWord_0) { .b1_0 = b1_2, .b2_0 = b2_2 };
}
```

```
static bool close_4(FILE* this, void* unused) {
  if (this != NULL)
    return fclose(this) == 0;
  else
    return true;
}


static void println_5(char* s_19) {
    print_4(s_19);
    println_6();
}

static int32_t getNextBufferBeginning_3(Dictionary_2* thiss_783) {
    if (thiss_783->nextIndex_1 == 0) {
        return 0;
    } else {
        return thiss_783->pteps_1.data[thiss_783->nextIndex_1 - 1];
    }
}


static bool isOpen_5(FILE* this) {
  return this != NULL;
}



static int8_t impl_4(FILE** this, void** unused, bool* valid) {
  int8_t x;
  *valid = fscanf(*this, "%c", &x) == 1;
  return x;
}



static void print_4(char* s) {
  printf("%s", s);
}


static void dropLast_3(Buffer_2* thiss_785) {
    thiss_785->length_3 = thiss_785->length_3 - 1;
}

static enum_Status_0 encodeFile_1(State* state_24) {
    FileInputStream input_1 = open_3("input.txt", *state_24);
    FileOutputStream encoded_1 = open_2("encoded.txt");
    bool norm_0 = isOpen_4(input_1);
    bool norm_2 = norm_0;
    bool norm_4;
    if (norm_2) {
```

```
    bool norm_1 = isOpen_5(encoded_1);
    bool norm_3 = norm_1;
    norm_4 = norm_3;
} else {
    norm_4 = false;
}
enum_Status_0 norm_68;
if (norm_4) {
    if (encode_4(input_1, encoded_1, *state_24)) {
        norm_68 = tag_Success_0;
    } else {
        norm_68 = tag_EncodeError_0;
    }
} else {
    norm_68 = tag_OpenError_0;
}
    enum_Status_0 res_9 = norm_68;
    close_5(encoded_1);
    close_4(input_1, *state_24);
    return res_9;
}

static int32_t getBufferSize_3(Dictionary_2* thiss_795, int32_t index_30) {
    if (index_30 == 0) {
        return thiss_795->pteps_1.data[0];
    } else {
        return thiss_795->pteps_1.data[index_30] - thiss_795->pteps_1.data[
     ↪index_30 - 1];
    }
}

static void* newState_2(void) { return NULL; }
```

C References

[1] "Leon Web Interface." http://leon.epfl.ch/.

[2] M. Antognini, "From Verified Functions to Safe C Code." https://github.com/mantognini/GenC, 2016.

[3] "clang: a C language family frontend for LLVM." http://clang.llvm.org/.

[4] "GCC, the GNU Compiler Collection." https://gcc.gnu.org/.

[5] X. Leroy, "A formally verified compiler back-end," *Journal of Automated Reasoning*, vol. 43, no. 4, pp. 363–446, 2009.

[6] "The Compcert C Compiler." http://compcert.inria.fr/.

[7] R. Blanc, V. Kuncak, E. Kneuss, and P. Suter, "An Overview of the Leon Verification System: Verification by Translation to Recursive Functions," in *Proceedings of the 4th Workshop on Scala*, SCALA '13, (New York, NY, USA), pp. 1:1–1:10, ACM, 2013.

[8] R. Blanc, "Verification by Reduction to Functional Programs." 2017.

[9] T. Welch, "High speed data compression and decompression apparatus and method," Dec. 10 1985. US Patent 4,558,302.

[10] R. Cohn, "A transpiler to convert Java to C source." https://github.com/raphaelcohn/java2c, 2015.

[11] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-C: A Software Analysis Perspective," in *Proceedings of the 10th International Conference on Software Engineering and Formal Methods*, SEFM'12, (Berlin, Heidelberg), pp. 233–247, Springer-Verlag, 2012.

[12] P. Baudin, J.-C. Filliatre, C. Marché, B. Monate, Y. Moy, and V. Prevosto, "ACSL: ANSI/ISO C specification language." http://frama-c.com/acsl.html, 2013.

[13] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and Understanding Bugs in C Compilers," *SIGPLAN Not.*, vol. 46, pp. 283–294, June 2011.

[14] N. G. Leveson and C. S. Turner, "An Investigation of the Therac-25 Accidents," *Computer*, vol. 26, pp. 18–41, July 1993.

[15] J. Garrett, J. Valenzuela, B. Douville, M. Carr-Robb-John, J. Adamczewsk, E. J. Douglas, R. Morwood, B. Burbank, and C. Pruett, "Dirty Game Development Tricks." http://www.gamasutra.com/view/feature/194772/dirty_game_development_tricks.php, 2013.

[16] "About the security content of macOS Sierra 10.12.1, Security Update 2016-002 El Capitan, and Security Update 2016-006 Yosemite." https://support.apple.com/en-us/HT207275, 2016.

[17] "Common Vulnerabilities and Exposures (CVE)." http://cve.mitre.org/, 1999.

[18] "CVE Details: The ultimate security vulnerability datasource." https://www.cvedetails.com/.

[19] D. Saks, "extern c: Talking to C Programmers about C++," *CppCon*, 2016.

[20] "Programming languages – C," standard, International Organization for Standardization, 1999.

[21] J. Gosling, B. Joy, G. L. Steele, G. Bracha, and A. Buckley, *The Java Language Specification, Java SE 8 Edition.* Oracle, 2015.

[22] *MISRA C:2012, Guidelines for the use of the C language in critical systems.* MIRA Limited, Nuneaton, 2013.

[23] "Heap fragmentation or how my micro-benchmark went wrong blog logo." `http://david-grs.github.io/heap_fragmentation_micro_benchmark/`, 2016.

[24] J. Regehr, "`INT_MIN % -1 = ?`." `http://blog.regehr.org/archives/175`, 2010.

[25] "`java.lang.Math`." `http://docs.oracle.com/javase/8/docs/api/java/lang/Math.html`.

[26] R. Madhavan and V. Kuncak, *Symbolic Resource Bound Inference for Functional Programs*, pp. 762–778. Cham: Springer International Publishing, 2014.

[27] "GCC: C Implementation-Defined Behavior." `https://gcc.gnu.org/onlinedocs/gcc/C-Implementation.html#C-Implementation`.

[28] "Visual Studio: Implementation-Defined Behavior." `https://msdn.microsoft.com/en-us/library/3d61ah0s.aspx`.

[29] "Clang: Bug 11272 - document implementation-defined behavior." `https://llvm.org/bugs/show_bug.cgi?id=11272`.

[30] M. Spencer, "My Little Optimizer: Undefined Behavior is Magic," *CppCon*, 2016.

[31] "Autoboxing and Unboxing." `https://docs.oracle.com/javase/tutorial/java/data/autoboxing.html`.