Can we Verify C with Whiley?

David J. Pearce
Victoria University of Wellington, Wellington, New Zealand

1 Introduction

The Whiley programming language has been developed from the ground up to enable compile-time verification of its programs [1, 2, 3, 4]. The Whiley Compiler (WyC) attempts to ensure that all functions in a program meet their specifications. When it succeeds in this endeavour, we know that: firstly, all function post-conditions are met (assuming their pre-conditions held on entry); secondly, all invocations meet their respective function's pre-condition; finally, that runtime errors such as divide-by-zero, out-of-bounds accesses and null-pointer dereferences are impossible. Note, however, that such programs may still loop indefinitely and/or exhaust available resources (e.g. RAM).

An interesting question for us is *whether or not Whiley can be useful in verifying legacy software such as that written in the C programming language*. This could be useful, for example, for finding zero-day exploits or other common errors (e.g. buffer overruns). The benefit of this approach is that the verification system can build on top of a language with clear semantics, but still be used to verify programs written in an inherently unsafe language (such as C). As an example, let us consider representing the C strings in Whiley. This is useful as we can then try to verify properties about functions operating on C strings (e.g. `strlen()`, `strcpy()`, etc). The main points about C strings are: **a)** they are arrays of 8bit ASCII characters (roughly speaking); **b)** they are terminated by the special NULL character '`\0`'; **c)** they do not carry any other length information (e.g. for the size of the containing memory chunk).

The interesting thing about Whiley is that we can encode these constraints within the language itself. Specifically, we're going to encode a C string as an array of integers with appropriate invariants. The array will be constrained to ensure it is null terminated, whilst the contained integers will be constrained to ensure they are between 0 and 255. Before giving our definition of a C string, we first need to define the notion of an ASCII character as follows:

```
type ASCII_char is (int n) where 0<=n && n<=255
```

We have defined an ASCII character to be an integer which is constrained between 0 and 255 (i.e. 8bits ASCII). Using this, we define our notion of a C string as follows.

```
type C_string is (ASCII_char[] str)
where |str| > 0 && some { i in 0 .. |str| | str[i] == NULL }
```

Here, a `C_String` is an array of integers constrained to ensure it is always null terminated. Using this, we implement the well-known `strlen()` function:

```
function strlen(C_string str) → (int r)
ensures str[r] == NULL && all { k in 0 .. r | str[k] != NULL }:
    //
    int i = 0
    //
    while str[i] != NULL
        where i >= 0 && i < |str| && all { k in 0 .. i | str[k] != NULL}:
        i = i + 1
    //
    return i
```

The Whiley compiler statically verifies this function does not overrun the string bounds. The loop invariant given by the **`where`** clause is needed as a hint to the verifier, but does not affect the function's execution in any way.

References

[1] The Whiley Programming Language, http://whiley.org.

[2] D. J. Pearce and J. Noble. Implementing a language with flow-sensitive and structural typing on the JVM. *ENTCS*, 279(1):47–59, 2011.

[3] D. J. Pearce and L. Groves. Whiley: a platform for research in software verification. In *Proc. SLE*, pages 238–248, 2013.

[4] D. J. Pearce and Lindsay Groves. Reflections on verifying software with Whiley. In *Proc. FTSCS*, pages 142–159, 2013.