# Twiddle – A DSL for the Functional Bit-hacker

Michael Buch

January 6, 2019

**Abstract**

It is useful (and fun!) to bit twiddle i.e. perform arithmetic and manipulate data at the granularity of individual bits. Traditionally there is a compromise one has to make between using a low-level unsafe language that allows bit-twiddling versus a safe high-level language in which the type system or language design prohibit operations at bit-level (without additional complexity). The *Twiddle* domain-specific language (DSL) is an embedded language written in Scala that generates bit-twiddling style C code. The language offers several modes of operation, useful for quick prototyping, debugging and custom extensions: (1) Tracing interpreter (2) Scala evaluator (3) Twiddle AST generator (4) C code generator. Thus our language is a tool for the curious, a tool for safe bit-hackers and a tool for someone looking to get a bit more performance out of his high-level language.

## 1 Motivation

Bit-twiddling offers performance, especially when inquiring about or operating on the bit pattern itself. However, reasoning about the final code is a daunting task and despite the fun-factor utility it can provide, programmers actually requiring to use such an algorithm would benefit from a verified solution built-into their language. The need for bit-twiddling most commonly arises in applications that actually care about the binary representation of an integer such as error-correction schemes. A programmer in such a situation essentially has two choices: (1) write an application in a low-level language like C and implement the readily available bit-hacks in it (2) use a higher level language and accept the overheads it incurs. This project proposes an alternative in which we code-generate low-level code from a safer high-level language meanwhile preserving bit-twiddling operations. We embed a miniature interpreted language within Scala, that permits evaluation but also code generation of C code. Thus a programmer can focus on the problem rather than the implementation details. One benefits from Scala's type-system and the the object language's expressiveness and pointer-freeness while being able to generate generate bithacks. Additionally, a user of said generated code can be reassured that the semantics of the bithacks implemented in the "Twiddle" language have been verified by the authors [1]. However, verifying the actual generated Twiddle code is not yet supported but could be realized in a similar fashion to Amin et al.'s generative verification framework [2].

## 2 The Language & Architecture

### 2.1 Core

The core of the language is split into modular pieces of functionality implemented as traits. As is common practice with tagless final interpreters (see section 4.4) we parameterize each set of language features with an evaluator that describes how each feature within its context. Below we briefly describe some of the core feature set which are implemented as traits in **core.scala**:

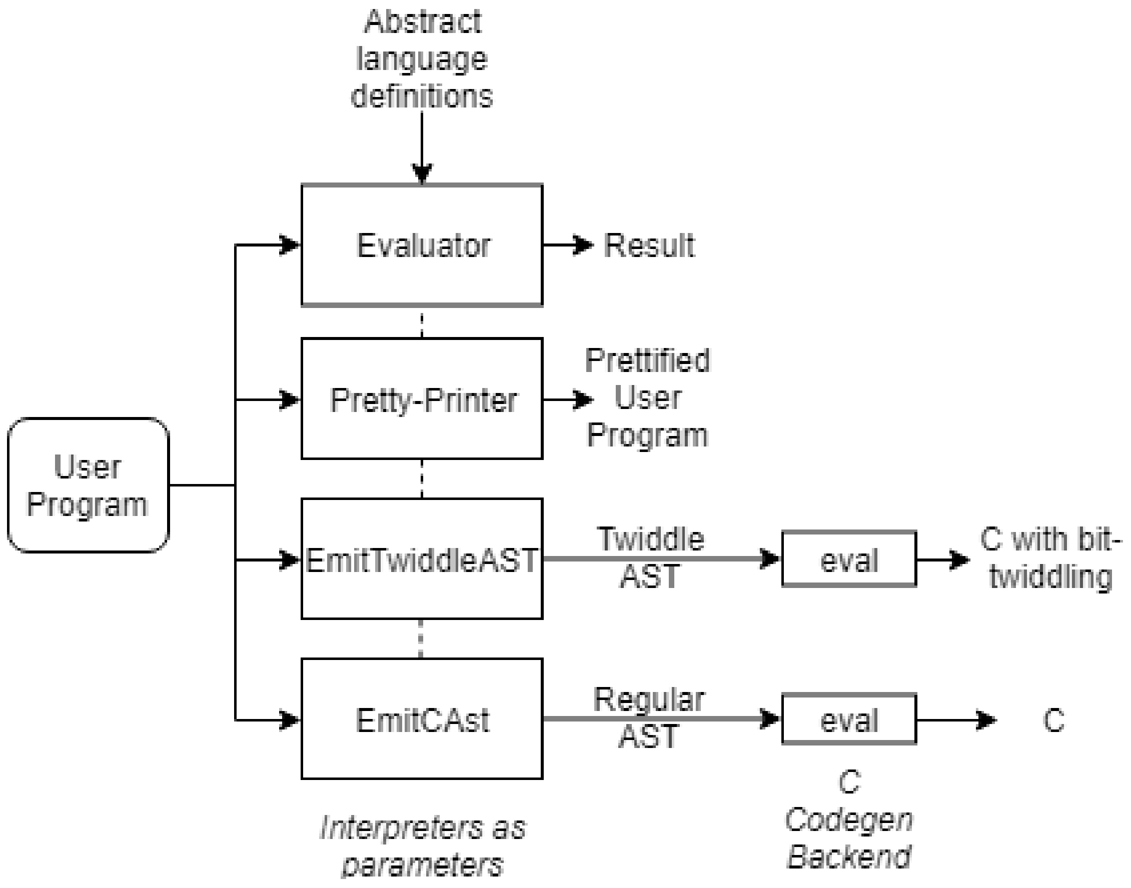

Figure 1: Architecture of the Twiddle framework

- **Nums**: Defines abstract interpretation of numeric Scala types including `Int, Double, Float`. It is generic over the type of input number supplied. Operations involving numeric types in the core language thus have to also be generic over numeric types. This is fine for integral numbers but gets problematic for fractional operations such as division.
- **Arithmetic**: Basic mathematical operations that operate on the interpretation of core's `num`. Some operations are generic over the input type while others are explicitly for fractional versus integral types.
- **Bools**: Includes both the interpretation of Scala's `Boolean` type and also of constructs operating on said booleans such as if-statements, ternary operators and logical operators.
- **Lambda**: Interpretation of lambda definition and application. Scala's lambda maps cleanly to the concept of a lambda in the lisp-like user-level language. However, a mapping to C is less straightforward and described in section 3.2
- **LispLike**: Contains a limited set of Lisp primitives to be able to create lists using `cons/car/cdr`. A `begin` primitive was added mainly as a way to collect the AST nodes created in a code snippet when interpreting

using `EmitTwiddleAST` since without it Scala would simply return the last generated statement. Essentially it is a wrapper around a Scala `List`.

The traits prefixed with a "C" such as the following ones listed are the operations taken from Anderson's set of bithacks[1].

- **CMathOps**: Mathematical operations that when passed to the codegeneration backend produce bit-twiddled implementations. Typically in other languages these would correspond to standard library functions such as **scala.math.log10** for `log10`.
- **Bits**: Includes definitions of the `bits` term, which is simply the interpretation of a numeric type, and functions whose goal is to operate on the structure of bit patterns of an integer such as reversing bit sequences.

### 2.2 Evaluators

Each term in the language is a function whose parameters and return type are wrapped in the abstract type of the interpreter that is evaluating the function. The second column of figure 1 shows the current set of supported interpreters.

*Evaluator* corresponds to the `implicit object Eval` in **interpreter.scala**. Here terms are evaluated and returned as values in the host language (i.e. Scala).

*Pretty-Printer* corresponds to the `implicit object Show` in **interpreter.scala**. Here all terms are of type `String` and evaluation of a term yields its pretty-printed version.

*EmitTwiddleAST* corresponds to the Scala object of the same name in **codegen.scala**. This is the core of the Twiddle code generator.

## 3 Codegen

### 3.1 Twiddle AST

The code generation part of the Twiddle framework is shown in the bottom two branches of the flow chart in figure 1. As a way to map modularly and extensibly between a high-level language like the lambda calculus variant in our case to a low-level language like C we introduced a intermediate representation (IR) of the user-level Twiddle code structured as an abstract syntax tree (AST). The IR language includes features that would not fit to the semantics of the high-level language but are needed for bit-twiddling or general operations in C. Nodes in the AST are all represented by the `abstract trait Term` and evaluation of the terms occurs on nested tuples of terms (i.e. `Tup(Term, Tup(Tup(Term, Term)...)...)`) to be able to conveniently manipulate it from within the evaluator of the AST. Where there is only a small set of one-to-one mappings between terms in the object language and actual C language constructs, the IR mimics a subset of valid terms in C. Figure DIAGRAM NEEDED shows valid terms in the Twiddle IR. Distinction between primitives, expressions, C language identifiers, etc. is not provided since our evaluators on the AST did not require it. The term types can, however, be refined with additional traits if one needs to special-case according to certain classes of C language features.

#### 3.1.1 Builtins

There is no distinction between language features and built-in functions in Twiddle. Every language feature is a function that describes its evaluation in the interpreter context. We implement common operations in the associated traits and inject it into the abstract `Exp[T[_]]` trait. These operations have been chosen with bit-twiddling in mind.

More often than not there is a corresponding algorithm using bithacks CITATION NEEDED which the codegen back-end implements in the IR.

We considered several ways of supporting bit-twiddling operations. In the final version of Twiddle we have a specific bit-twiddling backend whose sole purpose is to translate arithmetic, string or bit operations, provided as language features, into equivalent bithacks CITATION NEEDED. While a step in the right direction, an alternative (or extension) would be to detect certain user patterns or even annotations in the source, and map the patterns into bithack patterns. An example could be the reversal of a string. If currently a user is not aware of the Twiddle provided builtin `reverse(s: T[String])`, a user might choose to implement his own reversal algorithm and Twiddle would simply generate plain C. A more advanced backend detects these patterns from the AST.

#### 3.1.2 Bits

So far we have only discussed mathematical or string operations that utilize arithmetic or logical shift operators to achieve a task that would have required control flow constructs in a naive implementation. The term bit-twiddling, however, also encompasses operations directly on the bit representation of terms. For this purpose, we implemented the `bits(a: T[Int]): T[BitSet])` term in the core language. As the function definition implies, an evaluation of `bits` will turn an integer into a logical representation of bits. We say logical representation because this translation is different between the Twiddle front-end evaluators and the codegen IR. In the Twiddle AST bits are simply represented as integers. Operations on said integers that would in the C layer operate on bits are responsible for declaring the necessary integers as `unsigned long`. In the front-end evaluator, however, a conversion does take place from integer to a Scala BitSet.

#### 3.1.3 Other Useful Constructs

As implied above, not all IR terms are features in the core language, thus terms such as `Ref(e: Term)` or `Assign(v: Term, e: Term)` are helper terms used to ease the transition from user-level language to C. Variable declarations and assignment

## 3.2 Twiddle AST Interpreters

A consequence of the Tagless style of the core language is that in order to achieve a generic expression type that is used across all interpreters and be able to interpret the same piece of code in multiple ways, we need to implement language features across several interpreters that are semantically sound. For the `EmitTwiddleAST` interpreter this poses a challenge since some of the functional higher-level constructs do not map cleanly onto C-style programs. A prime example is the lambda/apply pair. A way to map lambdas to C is to convert the lambda into a function definition, storing the function body together with a unique function name in the internal interpreter state, return the function name to the user and on applications of the function name generate a C function call. For this purpose we implemented term `Func( FILL )` and `App( FILL )`. However, retrieving the function body requires intrusive changes around several components of the framework and is left as an open exercise for future versions. Currently a `lam` returns a closure to the user which he is responsible for applying i.e. passing to `app` which then evaluates the lambda.

#### 3.2.1 Auxiliary Machinery

To avoid name clashes we use a file-static counter that increments whenever a call to `gensym` is performed.

In procedural languages one has to be able to assign an expression to a variable to track state and perform any kind of meaningful computation. This is problematic in our “greedy-print” model of code generation since we can generate syntactically invalid expression assignments. An example would be generating code from:

```
val x = 5
val y = add10(x)
val z = 2 * y
```

```
int z = 2 * int x = 5; + 10 // Syntax error
```

Since we do not perform constant propagation or evaluation of terms prior to emitting Twiddle source, we need a mechanism to signal that an expression returns a meaningful result and a subsequent expression should use that value instead. Taking inspiration from Let-insertion in ANF-conversion of functional languages, we introduced a `Result(v: Var, t: Term)` into the IR that will assign the result of evaluating some expression $t$ to the variable $v$ in the generated code.

To overcome some of Scala’s type parameter constraints without adding unnecessary complexity, we introduce a dependency on the *shapeless* framework [1] which permits casting on type parameters even in the presence of type erasure. The main use is the casting of numeric types to a Scala `Long` when passing them to the `bits` evaluation.

```
val a = bits(num(20))
(ifThen(hasZero(a))
   (() => swapBits(a, bits(log10(num(30))))))
```

```
// <--- Snipped --->
#define HAS_ZERO(v) (((v) - 0x01010101UL) & ~(v) & 0x80808080UL)
#define SWAP(a, b) (((a) ^= (b)), ((b) ^= (a)), ((a) ^= (b)))
int main() {

int cond35;
unsigned int cons36;
unsigned int r34;
unsigned int v33;
v33 = 20;
r34 = HAS_ZERO(v33);
cond35 = r34;

if(cond35) {
     unsigned int r40;
     unsigned int v38;
     unsigned int v39;
     v38 = 20;
     int ret37;
     ret37 = (30.0 >= 1000000000) ? 9 : (30.0 >= 100000000) ? 8 : (30.0 >= 10000000) ? 7 : (30.0 >=
         1000000) ? 6 : (30.0 >= 100000) ? 5 : (30.0 >= 10000) ? 4 : (30.0 >= 1000) ? 3 : (30.0 >= 100)
         ? 2 : (30.0 >= 10) ? 1 : 0;
```

[1] https://github.com/milessabin/shapeless

```
    v39 = ret37;
    r40 = SWAP(v38,v39);
    cons36 = r40;
  }
  return 0;
}
// END OF CODE GENERATED BY TWIDDLE
```

Funcdefs IO Ensuring correct syntax

## 4 Design Choices

### 4.1 Lessons Learned

Value of tagless final alternative methods of emitting C->should probably follow semantics of the codegenned language

### 4.2 Language Embedding

### 4.3 Scala

### 4.4 Tagless Final Style

## 5 Conclusion & Future Work

- Benefits of shallow embeddings. Can fall back to scala features. Don't need to implement side effects in object language necessarily.
- Extend LMS
- Use tagless evaluator for AST evaluation
- Writing larger programs
- Other backends
- Staging

## References

[1] S. E. Anderson, "Bit twiddling hacks," *URL: http://graphics. stanford. edu/~ seander/bithacks. html*, 2005.

[2] N. Amin and T. Rompf, "Lms-verify: Abstraction without regret for verified systems programming," in *ACM SIGPLAN Notices*, vol. 52, no. 1. ACM, 2017, pp. 859–873.