Twiddle – A DSL for the Functional Bit-hacker

Michael Buch – mb2244 – Queens' College

https://www.github.com/Michael137/twiddle

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 parallelizer and 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 benefits, especially when inquiring about or operating on bit patterns themselves. However, reasoning about the final code is a daunting task and despite the fun-factor utility it can provide, programmers in actual need to implement fast versions of their algorithms using bit-tricks, 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 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 object language's expressiveness 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 and/or tested 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]. The domain specific language (DSL) incorporated into Twiddle is written in a tagless style, meaning it support various kinds of interpretations and is extensible to more. We provide the ability to both generate bit-twiddled and OpenMP annotated C code but also to evaluate or prettify the DSL.

1.1 Example

We start with an innocent looking snippet from the user-level language, a mixture between Scala (the variable assignment) and a simple-term language (the if-then construct):

```
def snippet[T[_]](s:Exp[T]) = {
   import s._
   val a = num(31)
   (ifThen(hasZero(bits(a)))
      (() => prints(quote("%d"), mod(bits(log10(num(1234567))), a))))
}
```

The type parameter `T[_]` represents the type wrapper that the evaluator `s: Exp[T]` uses to wrap its abstract type. We invoke the snippet as follows:

```
val evaluated = snippet(Eval)
val ast = snippet(EmitTwiddleAST)
```

Here we see the interpretation of the same snippet in two wildly different forms. The first line evaluates directly and yields the value "6". The second line, however, emits the AST using the `EmitTwiddlAST` evaluator, yielding:

```
Tup(Decl(I(Var(cond10)))),
Tup(Decl(U(Var(cons11)))),
Tup(Assign(Var(cond10),Result(Var(r9))
<--- snipped -->
```

When we pass the AST output to the code generator, i.e. `Codegen.gensrc(snippet(EmitTwiddleAST))` we get the following C code (with stylistic modifications):

```c
// <--- Snipped --->
#define HAS_ZERO(v) (((v) - 0x01010101UL) & ~(v) & 0x80808080UL)
int main() {
    int cond10;
    unsigned int cons11;
    unsigned int r9;
    unsigned int v8;
    v8 = 31;
    r9 = HAS_ZERO(v8);
    cond10 = r9;

    if(cond10) {
        unsigned int n13;
        unsigned int d14;
        unsigned int m15;
        int ret12;
        ret12 = (1234567.0 >= 1000000000) ? 9 : (1234567.0 >= 100000000) ? 8 : (1234567.0 >= 10000000) ?
            7 : (1234567.0 >= 1000000) ? 6 : (1234567.0 >= 100000) ? 5 : (1234567.0 >= 10000) ? 4 :
            (1234567.0 >= 1000) ? 3 : (1234567.0 >= 100) ? 2 : (1234567.0 >= 10) ? 1 : 0;
        n13 = ret12;
        d14 = 31;
```

```
    for(m15 = n13;n13 > d14;n13 = m15)
    {
        for(m15 = 0;n13;n13 >>= 5)
        {
            m15 = m15 + (n13 & d14);
        }
    }
    m15 = (m15 == 0) ? 0 : m15;
    int tmp16;
    tmp16 = m15;
    printf("%d", tmp16);
    cons11 = tmp16;
    }
    return 0;
}
// END OF CODE GENERATED BY TWIDDLE
```

This demonstrates our goal of producing bit-twiddled code from a safer high-level language without having to worry about syntax, details of bit representation, operator precedence or undefined behaviour. In the example the resulting value can be retrieved from variable `cons11`.

Another detail to note is the possible optimizations this enables by using the host language to resolve compile time decisions for the purposes of optimizing the generated code. The reason the modulus call, `mod`, was transformed to a loop as seen in the generated code above is that its value was 1 less than a power of two and enabled a known optimization that removes any division instructions. Had it been `32`, we would have generated a simple binary-and. Had it been just another integer we would have resorted to using the C modulo operator.

2 Run Instructions

2.1 Running Tests

The project structure is adapted to the Scala "sbt" build tool. The **src/main/scala/twiddle-tests.scala** contains feature test cases that showcase some of the functionality of Twiddle. To run them simply run following from the the root directory: `sbt "runMain twiddle.dsl.Main"`

2.2 "Twiddle Me This" REPL

An alternative mode of operation is the REPL included with the framework.

1. sbt "runMain twiddle.dsl.REPL"

2. Follow instructions in terminal

2.2.1 Example REPL session

```
$>>sbt ~"runMain twiddle.dsl.REPL"
#### Starting Twiddle REPL in twiddle mode ####
#### Other options are: 'eval','parallel','print','twiddle' ####
#### Toggle code generation mode on/off with 'c'/'!c'
#### Press 'q' to quit

>>> twiddle
==> Mode switched to twiddle

>>> c
==> Code generation mode *ON*

>>> log2(num(15))
float x1;
int ret2;
x1 = 15.0;
ret2 = *((const int* )&(x1));
ret2 = ((ret2 >> 23) - 127);

>>> eval
==> Mode switched to eval

>>> log2(num(15))
3.9068905956085187

>>> print
==> Mode switched to print

>>> log2(num(15))
log10(15.0)/log10(2)
```

   This is another useful demonstration of the power behind being able to interpret a language in multiple ways
with only a single core language definition. Users can switch between twiddling, parallelizing, evaluating, printing
or generating code from within the same REPL session.

2.3 Creating Snippets

1. Create a new file or use **src/main/scala/playground.scala**

2. Create a function with following structure:

   ```
   def example[T[_]](s:Exp[T]) = {
       import s._
       // Twiddle code goes here...
   }
   ```

3. Then call the function e.g. from a main() function, using one of the supported evaluators. When using `EmitTwiddleAST` you have several options of how to treat the AST currently:

- `Codegen.eval(example(EmitTwiddlAST))`
- `Codegen.gensrc(example(EmitTwiddlAST))`
- `Codegen.runsrc(Codegen.gensrc(example(EmitTwiddlAST)))`

The first will simply emit the Twiddled C code to stdout. The second method will generate a file called **twiddle.c** that contains the emittted source. The third option shows how to run the emitted source. `Codegen.runsrc(...)` will compile and run **twiddle.c**. `EmitTwiddleAST` can also be replaced by an equivalent AST generator such as `EmitParallelAST`.

4. Run from the root directory: sbt "runMain twiddle.dsl.Playground"

3 The Language & Architecture

3.1 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 5.2) we create an abstract type `Exp[T[_]]` parameterized by an evaluator. Each set of language features is then implemented as a function in an evaluator that describes the feature's behaviour in the evaluator's context. Simply said, we turn terms in the host language into functions in the object language. We bundle similar language features into a set of traits. Some of these features (defined in **core.scala**) are described below:

- **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 4.2

- **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`.

Some traits bundle operations taken from Anderson's set of bithacks[1]:

- **CMathOps**: Mathematical operations that when passed to the code-generation backend produce bit-twiddled implementations. Typically in other languages these would correspond to standard library functions such as **scala.math.log10** for `log10`.
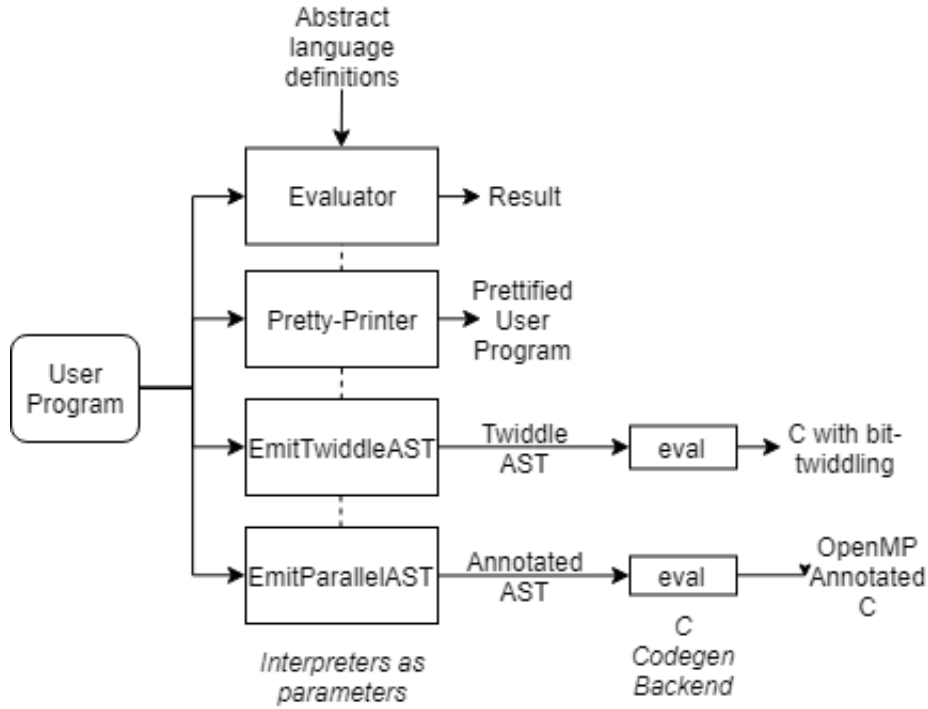
5

Figure 1: Architecture of the Twiddle framework

- **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.

3.2 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 **ast-interpreter.scala**. This is the core of the Twiddle code generator.

EmitParallelAST corresponds to the `implicit object EmitParallelAST` in **ast-interpreter.scala**. This evaluator produces an AST with annotations for parallelization. The code generator uses the annotations to generate additional OpenMP pragmas.

4 Codegen

4.1 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 such as our simple term-language 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)...)...)`, in order 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. This architecture provides an easier transition from host language to the target and opens the opportunity for additional optimizations on the IR before code is generated.

The actual code generation is a simple case-based evaluator that matches on nodes in the AST. This decouples the DSL language evaluation from the emission of code and proved to be convenient to work with when adding more features into the language.

4.1.1 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 [1] 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. 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.

4.1.2 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 int`. In the front-end evaluator, however, a conversion does take place from integer to a Scala BitSet.

4.1.3 Other Useful Constructs

As implied above, not all IR terms are features in the core language, thus terms such as `Ref(e: Term)`, `Decl(e: Term)` or `Assign(v: Term, e: Term)` are helper terms used to ease the transition from user-level language to C. Some of the helper terms have fine-grained alternatives to match the possible syntactic uses of statements in C. For example, assignment is also supported via the AST node `AssignInline(...)`, which drops the semicolon and linefeed from a regular `Assign(...)`.

4.2 Twiddle AST Interpreters

A consequence of the Tagless style of the core language is that in order to achieve a generic expression type one has to make sure features can be used across several interpreters while remaining 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(...)` and `App(...)`. 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.

4.2.1 Auxiliary Machinery

To avoid name clashes in generated variables 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
```

to

```
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 [3], we introduced a term, `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 make sure the C preprocessor directives such as `#include` and `#define` are kept at the top of the file (although generally speaking C programmers might not want this), we keep a file global list of defines together with metainformation about them, such as name, paramters and body. We collect the information during AST evaluation and emit them during code generation (in `Codegen.gensrc(...)`).

5 Design Choices

5.1 Scala

We chose Scala primarily for following reasons:

- Functional features that benefit implementation complexity

- Flexible type sytem

- Reflection capabilities

- Library support for staging

The combination of Scala's ability to create type classes, pattern matching, first-class functions and mature reflection capabilities are a well-suited foundation for writing an interpreter or embed a DSL into it. We make use of type classes [4] to implement our tagless interpreter by having traits that represent the DSL's language features be defined for multiple implementations. Where we face issues with implementing functionality for individual interpreters, for example due to restrictions imposed by that interpreters abstract type (or sub-optimal design decisions), we use reflection in the form of `ClassTag` or `TypeTag`. Our code generation backend is essentially a pattern matcher on terms.

The last point is in regards to Rompf et al.'s Leightweight Modular Staging framework [5] that uses type-driven staging to perform automatic binding time analysis and program specialization. It is attractive due to its extensibility to ones own DSL, providing multiple code generation targets including C and CUDA, which would have been ideal for Twiddle. Unfortunately, practical limitations including the lack of flexibility around the amount, name and properties of generated variables (without significant implementation overhead) prevented us to fully committing to its use. Additionally, because LMS shares code generation facilities between C++ and C, finding the correct combination of traits to mix into the Twiddle DSL to generate *standard* C without creeping C++ into the mix proved challenging. After initial implementations using LMS we instead dropped support for the framework and built our own code generation facilities. However, since other DSLs targeting C have been implemented using LMS successfully [6, 2], a future iteration of Twiddle could have its code generation and optimization be done through LMS.

Of course similar features are avaiable in languages like Haskell and Ocaml and staging/metaprogramming facilities in frameworks like Template Haskell and MetaOcaml. However, due to experience prior with Scala and its mature reflection API that is part of the *language standard* served as an advantage over the alternatives.

5.2 Tagless Final Style Embeddings

While the core language is implemented as a tagless-final interpreter, our code generation backened uses a tagged approach. An advantage of a tagless approach is that encourages to carefully think about the type signatures of the core language features. Since a term's interpretation can vastly differ between evaluators, a poorly chosen API can prove difficult to implement such that it works across all evaluation schemes.

An alternative to our case-based code generation could have been to implement the code generation as just another tagless interpreter and skip the creation of an AST. Since C is a language of vastly different structure and semantics than both the host language and the user-level language, a separation between front-end evaluation and code-generation benefits the extensibility of the framework. Essentially we can now imagine alternative AST generators, ones that produce an AST with nodes that are optimized for example with respect to code size instead of just being twiddled. Our design achieves a clear separation between analysis of the object language and the act of mere code-generation which can be reused for multiple forms of the AST.

One downside of our approach is that we basically have to model the entire C language's semantics in our AST to be practical.

6 Conclusion & Future Work

During the development of the framework we benefited from both the shallow embedding and tagless final style of our DSL. We fall back to many of Scala's feature during evaluation which reduces the complexity and feature size of our DSL significantly. It also means we do not necessarily have to implement side-effects into the object language but instead defer them to Scala. The tagless style provides a useful guide when adding new core features into a language and although it can be punishing to ill-designed language APIs, it provides a robust way to extend a language across multiple interpreters.

Current limits of the framework include the amount of supported language features, intersection of the supported types between the user-language and C and the number of evaluators avaiable. Additionally, much of the current DSL fits the structure that the Scala LMS framework expects to be able to add code-generation and optimization facilities to multiple backends.

Our language currently only supports basic string, integer and 32-bit bit types. C additionally supports various numeric unsigned/signed/numeric/fractional types of different sizes. Moreoever, we do not not support collection types such as arrays or structs (although they can be emitted through pointers via the AST). A natural improvement to Twiddle would be the extension of the core language to a larger conformance with the C language especially the ability to define and call functions.

A useful additional extension could be the verification of the output language. For this one could add annotations to the code that conform with available C verification backends such as Frama-C [7]. Another option would be to investigate extensions to existing verified-C generation frameworks such as "lms-verify" [2].

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.

[3] A. Sabry and M. Felleisen, "Reasoning about programs in continuation-passing style," *Lisp and symbolic computation*, vol. 6, no. 3-4, pp. 289–360, 1993.

[4] P. Wadler and S. Blott, "How to make ad-hoc polymorphism less ad hoc," in *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* ACM, 1989, pp. 60–76.

[5] T. Rompf and M. Odersky, "Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls," in *Acm Sigplan Notices*, vol. 46, no. 2. ACM, 2010, pp. 127–136.

[6] H. Lee, K. Brown, A. Sujeeth, H. Chafi, T. Rompf, M. Odersky, and K. Olukotun, "Implementing domain-specific languages for heterogeneous parallel computing," *Ieee Micro*, vol. 31, no. 5, pp. 42–53, 2011.

[7] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-c," in *International Conference on Software Engineering and Formal Methods.* Springer, 2012, pp. 233–247.