Twiddle – A DSL for the Functional Bit-hacker

Michael Buch

January 3, 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

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 4.3) we parameterize each set of language features with an evaluator that describes how each feature within its context. The core set of features is divided into following traits:

1. Arithmatic
2. Strings
3. Bools
4. Lambda
5. LispLike
6. CLike
7. CMathOps
8. CStrOps

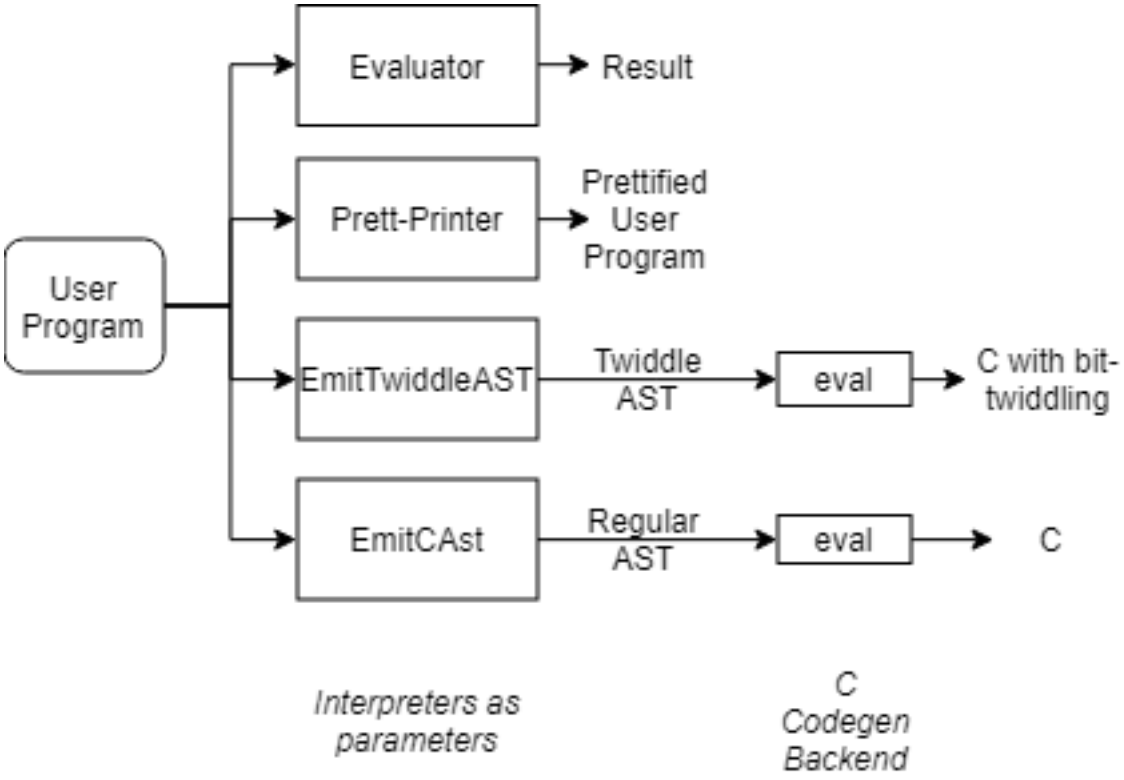

Figure 1: Architecture of the Twiddle framework

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.

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.

Arithmetic/Math operations Variable declarations and assignment Twiddle built in functions

3.2 Twiddle AST Interpreters

Gensym Return helper Lambda/apply IO

4 Design Choices

4.1 Language Embedding

4.2 Scala

4.3 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

References