Chapter 8
Operator Overloading as a DSL Parsing Mechanism

Alberto Simões
Polytechnic Institute of Cávado and Ave, Portugal

Rui Miguel da Costa Meira
Polytechnic Institute of Cávado and Ave, Portugal

ABSTRACT

This chapter describes an approach for the implementation of embedded domain-specific languages by using operator overloads and the creation of abstract syntax trees in run-time. Using the host language parser, an AST is created stating the structure of the DSL expression that is later analyzed, simplified, and optimized before the evaluation step. For the illustration of this process, the chapter proposes a domain-specific language for a basic linear algebra system dealing with matrices algebra and its optimization.

INTRODUCTION

Domain Specific Languages (DSL) are, undoubtedly, a great approach for the acceleration of processes (Kosar et al, 2015). These processes can be at different level: either for a specific purpose, completely outside of the scope of the programming paradigm (e.g. a specific language for describing a taxonomy of a thesaurus) or to help in the development process (e.g. the well-known flex and bison languages, designed to help in the development of compilers). While the first require a syntax specifically designed for that purpose, as the end users are not necessarily programmers, the

DOI: 10.4018/978-1-5225-7455-2.ch008

second usually take a hybrid approach, where some details are described in a new syntax, but a lot of the syntax is from the target language.

Usually DSL are classified as proper languages, when they are developed as a standard new language, where parsing follows the traditional approach (lexical and syntactic analysis, abstract syntax tree creation, tree manipulation and code generation or evaluation) or as an embedded language.

For this second situation, some DSL implement code generation, creating code in the target language that will be compiled and integrated with other host language files, or allowing its evaluation on run time (mainly for interpreted or other languages with reflection or meta-programming support (Bracha & Ungar, 2015)).

As parsing mechanism, this second case uses language constructs to define a dialect of the host language or, in some other situations, a hybrid parsing approach where some high order function transforms parts of the DSL syntax in the host language syntax (see (Simões & Almeida, 2010) for such a DSL implementation).

In this chapter we present another way for the development of embedded DSL through the use of operator overloading. While operator overloading is a common functionality on recent object-oriented languages, like C++, Java, C#, Python or Perl, the way these operators are used is, in most situations, the simple replacement of the default operator behavior (for example, the sum of two numeric values) with a similar one (for example, the sum of two vectors).

There are other situations where these operator overloads can create an abstract syntax tree (AST) instead of trying to evaluate the operator semantic. Thus, this behavior would be very similar to what a traditional parser would do when analysing the language. The main different is that it will be done in run time.

Note that this is not a new approach, as the way some libraries work show that similar approaches are used. An example is the way TensorFlow (Abadi et al, 2016) is able to compute what they call a computation graph, and later infer this computation derivative automatically. Therefore, this chapter does not claims a new methodology, but rather the clear definition of the structure of such a DSL implementation. Another example is MXNet (Chen et al, 2015) library. While the authors refer the usage of embedding a DSL in a host language, no references are made. Authors describe their systems results rather the way their implementation was done.

A few other references (Corliss & Griewank, 1993; Phipps & Pawlowski, 2012) were found, where the idea of operator overloading is discussed in order to enable differentiation and integration of expressions. But no details on DSL embedding are given.

MOTIVATION

The motivation for the approach here described is the construction of a basic Perl framework to act like TensorFlow, for simple neural networks, and easy to understand and edit by the community. While the module is under work and at a reasonable distance from being really useful, its development arose some problems that lead to the architecture here described.

The implementation of a neural network deals mainly with Basic Linear Algebra (BLA) operations. Given the requirement of fast implementations, these operations were obtained through a Basic Linear Algebra Subprograms (BLAS) library. The more relevant methods were defined: matrix sum and subtraction, element-wise operators (sum, multiplication, exponentiation, etc), matrices inner product, matrix transposition, matrix inverse, sigmoid and tanh, and some other operations.

To allow the users to easily write algebraic expressions with matrices, common operators were overloaded. The four arithmetic operators were overloaded for element-wise operators or matrix-matrix operator (if making sense) just by looking to the operator arguments' types. The special 'x' repetition operator existing in Perl was overloaded for the matrix inner product. Other than operators, some functions were defined for transposition, inverse, sigmoid, tanh, softmax and other relevant operations.

As these operators are guaranteed by the BLAS library, the first overload mechanism was to directly call the respective BLAS function for each required operation.

While working, this approach had some drawbacks:

- It was not possible to take advantage of specific optimizations present in the library. As an example, Lapack (Anderson et al, 1999) BLAS implementation allows the inner product between two matrices to be done with optional transpositions for any of the matrix arguments. If the evaluation is done at each operator or function call, when the expression A x B^T is parsed, first B would be transposed, and only then the inner product would be performed. Nevertheless, it is possible to do everything at once with BLAS.
- For neural networks and deep learning, some operations need to be done over and over again, with the same expression, replacing only one of the arguments. If every operation is performed as soon as it is parsed by Perl, they will be evaluated every time, even if a subexpression is not changed and its value can be reused.

- In the future, if there is the interest of computing derivatives automatically (this is not yet a priority in our work) we will need to store the computation graph (basically, an AST for the expression) to be able to obtain its derivative automatically.

These problems and goals were the main reason for applying the approach we describe in the next sections. As our development continues, the choice seems adequate and working as expected.

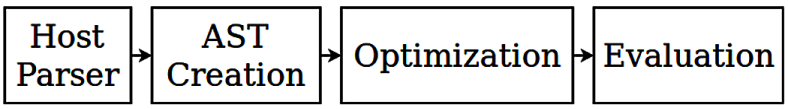
ARCHITECTURE

The common approach for the implementation of a programming language compiler or interpreter is comprised by the lexical and syntactic analysis part, the creation of an abstract syntax tree (AST), a possible step on optimization that rewrites the AST, and the AST traversal for code generation or execution.

In our proposed approach all these steps exist as well. The main difference is that the lexical and syntactic analysis is performed by the original host language parser (in this case, hitchhiking with the Perl parser). This means that the syntax of the defined DSL should be somehow a subset of the host language. While this can be a limitation, if the host language is properly chosen, it might not be a real problem. Figure 1 shows this architecture as a diagram.

In the next step, an abstract syntax tree should be created. This is performed by a set of methods, one for each overloaded operator or function, that returns an AST node. For a Object Oriented language these methods are very simple, gathering the objects returned by each operand, and saving a reference to them in the tree node. These are the methods that will be called by the overloaded operators, instead of the

Figure 1. Proposed architecture



common evaluation methods. Thus, the result of parsing an expression is a variable holding a tree of objects.

Follows the optimization step, adapting the AST accordingly with the language needs. Usually it can be implemented as a simply tree traversal, that change its structure where required. The main challenge is to understand when this step should be performed, and guarantee that, if performed once in a subtree, it is not performed again (in order to guarantee execution performance).

The final step is the evaluation or code generation. It is responsible for a final traversal of the tree. For interpreted languages, the traversal will evaluate through a depth-first tree traversal. First evaluating the leafs of the tree, and then applying each operator to the already evaluated children. Whenever possible, the implementation should take care of guaranteeing that the evaluation for a subtree is cached and not performed each time the expression value is needed (as long as its argument values are unchanged).

The next sections will describe each one of these three steps: the AST construction, the Optimization of the AST, and its Evaluation. While the explained ideas are not language dependent, the examples will be shown using Perl syntax, for the motivating DSL: a simple Basic Linear Algebra (BLA) expression evaluator.

AST CREATION

As the goal is to implement BLA expressions, AST leafs are scalar objects: matrices, numeric values, and placeholders (used for the user to replace a specific value in the computation graph). Internal nodes are standard objects. While Perl is not typed, and therefore, uses untyped variables, it is possible to query scalar values in order to know if they are basic types (as numeric values or strings) or objects, and for these detect their type (if they are matrices or internal nodes).

For the creation of matrices leafs (we consider a vector as a matrix where one dimension is 1) a call for the matrix constructor is needed. There are different constructors, for matrices of zero values, matrices where all values are the number 1, matrices of random values, identity matrices, or specific values matrices. All these

constructors have a shorthand method name, that is exported to the host language function table, and declared with the exact amount of parameters needed. This allows their usage with (or without) the need of syntactic sugar:

```
$zeros = Zeros 10, 20;
$ones = Ones 1, 50;
$random = Random 20, 20;
$identity = Identity 10;
$custom = M [[10, 20, 30], [40, 50, 60]];
```

Note that Perl allows that, for methods defined prior their usage with a specific signature, to discard the usual parenthesis used in function call. Thus, the only syntactic sugar required by the host language are the dollar signs in the variable names, and the semicolon at the end of each command. For other leaf types, like integer or real values, they are used in their host language format.

To create the AST nodes, as already mentioned, our approach uses operator overloading, and function creation or redefinition. Given the flexibility of Perl basic object system, the AST internal tree nodes are generic dictionaries, where each key is a field, and values are those fields values. This allows different nodes to have different structures without the need to define different classes.

For simple algebra operators (sum, subtraction, division and multiplication) the AST creation is a simple call to the constructor for a tree node structure, where the operator and arguments are supplied accordingly. As an example, follows the implementation for the method that is used to overload the sum operator:

```
sub sum ($left, $right) {
      return AST_Node(operator =>  'sum', args => [$left,
$right]);
}
```

Note that operator precedence table is shared with the precedence table of the host language operators. Thus, when overloading a specific binary operator, the host language is responsible for supplying the left and right operands to the overloading function. The overload for the 'x' operator is similar, as it is also a binary operator (although not much languages have it available).

For functions, like the computation of the transposed matrix, inverse, softmax, sigmoid, etc., the tree constructor is called from the called method. As an example, the T function is defined as:

```
sub T ($matrix) {
     return AST_Node(operator => 'transpose', args => [ $matrix
]);
}
```

While the user is calling a method to compute the transposed matrix, what is done really, is to store its AST, and do the required computation only when needed. Again, Perl allows the resulting DSL to recognize the following syntax:

```
$c = $a x T $b;
```

Indeed, this syntax is still close to the host language syntax. But it is clean and, to be used as an embedded DSL, acceptable, as it is both versatile enough, and similar with the host language.

For other functions, like trigonometric functions or the exponential or logarithmic functions, that already exist in the host language, a little more work is needed, as it does not make sense to create a function with a different name to be used in the DSL. Specially as it would result in different function names for numeric values (the built-in functions) and for matrices (our defined functions).

The solution was possible given Perl reflection and meta-programming. A reference to the original functions is fetched from the symbol table and stored with a different name. The new function is defined with the name of the built-in method, but returning the AST node for that operation. Later, in the evaluation function, the original method is called when the argument is a numeric value, and the matrices method call when dealing with a matrix.

OPTIMIZATION AND SIMPLIFICATION

The optimization or simplification step tries to find similar or related operations that can be performed more efficiently as a single one, than when executed independently. Examples are the inner product between two matrices, where one or both of them are transposed. BLAS implementations allow these different situations to be handled by the same method. Another situation is the sequence of similar operations where the temporary matrices created by each operator evaluation method can be reduced. These two examples are detailed below.

This task can be defined as a tree rewriting process, that detects tree patterns that can be simplified or optimized, and replaces them with the respective optimized version.

The approach implemented for our DSL is based on a depth-first traversal, starting to optimize leafs, their parents, and up the tree to the root node. This way each node optimization can query their children that are already optimized or simplified. Also, given that a subtree can be shared between two or more tree expressions, the optimization process adds a flag to each tree node stating the status of that subtree optimization.

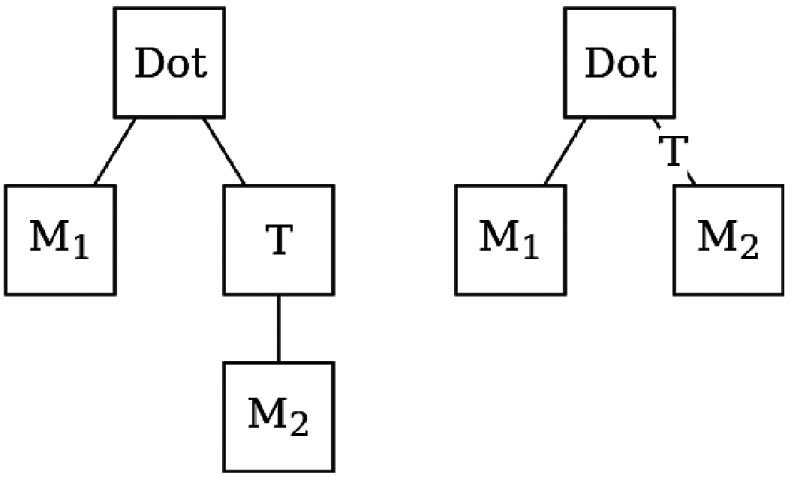
Also relevant is to refer that this optimization does not need to be performed at the end. It can be done at the same time as the AST tree is being built. Everytime a new node is added, the direct subtree can be analysed and the group of nodes simplified when possible.

Follows two specific examples of optimizations implemented in our BLA language:

1. First, the inner product where one of the matrices is transposed: $M_1 \times M_2^T$

While the transposition can be done first, and then the inner product, this process can be made faster as the usual implementation of the inner product allows the usage of the same code when one or the two matrices are transposed, as there is only a change of the order of the two nested cycles (for columns and rows). Figure 2 presents at the left the original expression, and at the right the optimized version, where the transposing information was synthesized by the main operator: $M_1 \, x^T \, M_2$.

Figure 2. Example of optimization for dot product



2. Second, a sequence of the same algebraic operator: $M_1 + M_2 + \ldots + M_n$

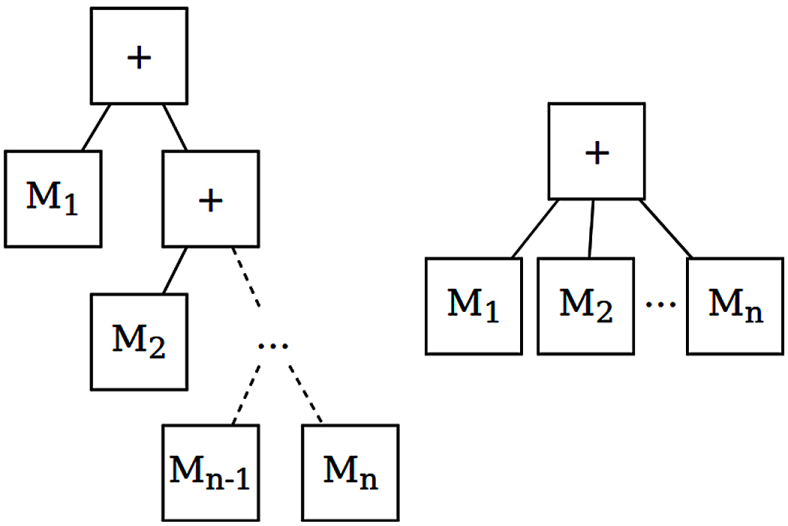
This expression AST is created like the example at the left of Figure 3. This means that, for a sequence of *k* matrices, *k-1* nodes are created. When each of these nodes are evaluated, a new temporary matrix is created to store the sum result. Thus, a total of *k-1* temporary matrices are needed. The optimization step can transform this tree into a single node AST, as presented at the right in Figure 3. This solution will create a new temporary matrix with the sum of the two first matrices, and them accumulate in that temporary matrix the value for the remaining sums. While the operation is repeated the same *k-1* times, the lack of a temporary matrix is enough to make the process both faster and less memory consuming.

EVALUATION

While the evaluation of an AST is not a problem, the main issue is to understand when this evaluation can be performed. This can highly depend on the host language and its functionalities regarding operator overloading and meta-programming availability.

The main problem is that the evaluation can not be done at the same time as the tree is constructed. For example, when adding a new node (say, a transposition operator), there is no information about its parent node type, and therefore, there is no enough information regarding if that node should be evaluated (and the transposed

Figure 3. Example of optimization for a sequence of n-1 sums



matrix computed) or if it should be kept intact, so that, later, its parent can take advantage of it, and optimize the complete evaluation.

Thus, the evaluation should be performed when there is the need to query the final value of the expression. For a matrix, when it is being printed, compared, or a specific element value is retrieved. For a computation that yields a numeric value, when that value is needed to be compared, or printed.

For Perl there are a few tricks that help this process. First, Perl includes two pseudo-operators that can be overloaded, namely 0+ and "". These operators refer, respectively, to a method to convert a variable into a value number, and to a method that converts the stored data into a string. These two methods are available to Perl as it does not distinguish data types, and whenever it needs to perform a computation with a value, it uses the 0+ overload to obtain the numeric value; and uses the "" overload whenever it needs to print that value. Thus, with these two specific operators we can infer that evaluation should take place whenever they are called.

Unfortunately there are other problems. Given a matrix is stored as an object, some methods (like the accessor for a value stored in a specific position of the matrix) require that the expression is evaluated, so that the matrix is available, and that specific position can be queried. One option could be adding code to each of these methods, in order to evaluate the expression before running its behavior. But this process is not easily scalable.

Fortunately, Perl allows the programmer to intersect calls to an object when there is not a method available with the referred name. For example, if a method "xpto" is called in an object that does not implement it, before complaining, perl calls a AUTOLOAD method, that can load dynamically methods from external libraries.

Our implementation uses this functionality. When an unknown method is called for an expression, the expression object can infer that the method should not be called on the expression itself (as it is not defined for that kind of objects), but on the result of its evaluation. With this in mind, the evaluation method is called once, the evaluation value cached, and the method called on the evaluation result (typically a matrix). While this works perfectly, there is some kind of indirection that is not possible to implement in all OO languages.

CONCLUSION

Using operator overload, as well as defining new functions or redefining functions from a host language is a simple way to embed some type of domain specific languages in a host language. While the majority of the host language structure is kept, it is possible to redefine the operators to make them act correctly for the specific arguments they receive.

While operator overloading is a common thing, most users do this as an evaluation step, meaning that the overload for a specific operator executes immediately the operation on their parameters. This approach works perfectly for basic languages, but is not versatille when there can be some kind of optimization or simplification of compound operations.

To help on this process, we describe how the operator overload and new defined methods can be used to create an abstract syntax tree (instead of direct evaluation) that can be then simplified or optimized as needed. Finally, whenever the real result of the expression is needed, the computation graph can be evaluated.

This is not a new approach, and is being used on some libraries. Nevertheless, our main contribution is the systematic description of the process as reference for other language designers who need similar functionalities for their projects.

One of the main advantages of this process is that, given operators and functions operate at the object level, one host language might embed more than one of these domain specific languages, given their overloads will not collide. This is a real advantage, as most embedding approaches usually require a parsing step which might interfere with the parsing step for a different embedded language.

REFERENCES

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., . . . Kudlur, M. (2016, November). Tensorflow: a system for large-scale machine learning. In OSDI (Vol. 16, pp. 265-283). Academic Press.

Anderson, E., Bai, Z., Bischof, C., Blackford, L. S., Demmel, J., Dongarra, J., & Sorensen, D. (1999). *LAPACK Users' guide*. Society for Industrial and Applied Mathematics. doi:10.1137/1.9780898719604

Bracha, G., & Ungar, D. (2015). OOPSLA 2004: mirrors: design principles for meta-level facilities of object-oriented programming languages. *ACM SIGPLAN Notices*, *50*(8), 35–48. doi:10.1145/2854695.2854699

Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., . . . Zhang, Z. (2015). *MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems*. Retrieved from http://arxiv.org/abs/1512.01274

Corliss, G. F., & Griewank, A. (1993). Operator overloading as an enabling technology for automatic differentiation (ANL/MCS/CP--79481). Academic Press.

Kosar, T., Bohra, S., & Mernik, M. (2015). Domain-Specific Languages: A Systematic Mapping Study. *Information and Software Technology*, *71*. doi:10.1016/j.infsof.2015.11.001

Phipps, E., & Pawlowski, R. (2012). Efficient Expression Templates for Operator Overloading-Based Automatic Differentiation. In Recent Advances in Algorithmic Differentiation (pp. 309-319). Academic Press.

Simões, A., & Almeida, J. J. (2010). Processing XML: a rewriting system approach. In XATA 2010 --- 8ª Conferência Nacional em XML. Aplicações e Tecnologias Associadas.