GRAMMAR PARSER-BASED SOLUTION FOR THE DESCRIPTION OF THE COMPUTATIONAL GRAPH FOR THE GNA FRAMEWORK

N.S. Tsegelnik[a], M.O. Gonchar, K.A. Treskov

Joint Institute for Nuclear Research, RU-141980 Dubna, Russia

E-mail: [a]tsegelnik@jinr.ru

The data flow paradigm has established itself as a powerful approach to machine learning. In fact, it is also very powerful for computational physics, although it is not used as much in the field. One of the complications is that physical models are much less homogeneous compared to ML, which makes their description a complicated task. In this paper we present a syntax analyzer for the GNA framework. The framework is designed to build mathematical models as lazy evaluated directed acyclic graphs. The syntax analyzer introduces a way for a concise description and configuration of the models using math-like syntax, providing scalability and branching. The goal of the project is to develop a technique and a software to facilitate a generic analysis and input data description compatible with multiple backends, e.g. GNA.

Keywords: grammar parser, syntax analyzer, data flow, high performance computing, GNA

Nikita Tsegelnik, Maxim Gonchar, Konstantin Treskov

1. GNA overview

GNA is a framework for fitting large-scale physical models [1], for example a Daya Bay model [2]. The framework uses the data flow concept [3] within which a model is represented by a directed acyclic graph. Each node is an operation on an array, e.g. matrix multiplication, derivative or cross section calculation, etc. GNA enables the user to create large-scale lazily evaluated models, handle large numbers of parameters, propagate parameters uncertainties while taking into account possible correlations between them, fit models, and perform statistical analysis.

The framework is implemented in C++ and Python: the core (backend) is implemented in C++ to provide efficiency while the management code (frontend) is written in Python to provide flexibility. GNA is currently under active development and we have several main goals ahead of us, e.g. advanced GPU support or automatic differentiation (AD). There are two ways to achieve success in this direction: to develop the current codebase further by implementing the AD and GPU support or to use an already existing solution.

The latter seems to be the best choice because it opens up new opportunities for the project development, such as using well-known and proven solutions with a large community, e.g. TensorFlow [4] or PyTorch [5], with which GNA has a lot of similarities. In this case, GNA will require the support for multiple backends. Another promising solution is the use of the high-performance programming language Julia [6], which not only has an easy-to-use and powerful GPU and AD support, but also initially implements a model for creating large parallel applications. At the moment it is not clear how much work will be required for porting the GNA models on either of the solutions. Moreover, the final performance improvement of the large-scale models is also uncertain. Thus, the transition requires careful research.

2. GNA usage

A physical model in GNA is represented by a computational graph, which consists of nodes — functions that operate on arrays of data. The data memory is allocated on the outputs of the nodes. The outputs are connected to the inputs of the other nodes forming a data flow scheme. Leaving the inner details behind the scenes, we are interested in nodes and data. For historical reasons the nodes have two representations, which in terms of GNA are called *transformations* and *variables*. Transformations typically operate on arrays of data. The inputs and outputs of transformations are explicitly connected within a Python code in order to build the models. The variables are used to represent scalar numbers, which may be modified by the minimizer. They are located in a namespace and are bound to the transformations via name search.

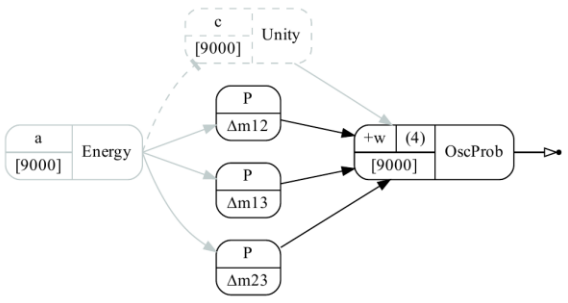
The GNA framework provides a library of transformations required for building physical models and performing statistical analysis of data. The included transformations consist of basic arithmetic (e.g. sum, product), linear algebra (e.g. Cholesky decomposition), statistics (e.g. covariance matrix, χ^2), calculus (e.g. differentiation, integration, interpolation), physics (e.g. probability of neutrino oscillations, inverse beta decay cross section), and detector effects (e.g. energy smearing, energy response distortion). There is a focus on reactor neutrino physics due to the fact that the project has grown from the analysis of Daya Bay experimental data. Currently GNA is used for the analysis of the data of the reactor and accelerator neutrino experiments.

To use any model within GNA the binding stage must be performed. At this stage, the transformations (and variables) are being instantiated and bound. While the binding step may not be efficient it is typically executed only once. After that procedure the model may be repeatedly used for computation. Since the framework guarantees that the inputs and outputs passed to the computation functions are of proper shape and type the corresponding checks are omitted in the computation functions, which makes them more readable and efficient.

One can perform a binding procedure using Python. The neutrino oscillation probability formula, example of the binding in GNA and resulting computational graph are shown in [fig. 1]. Nodes are only evaluated if they have changed and the result is cached and reused. The variables are maintained in a recursive namespace (*ns* in the example). Within this approach the nodes do not own variables they depend upon.

```
E = C.Points(np.arange(1.0, 10.0, 0.001))
with ns:
  oscprob = C.OscProb3(from_nu, to_nu)
  unity = C.FillLike(1)
  ws = C.WeightedSum(weights, labels)

  E >> unity.fill
  E >> oscprob.comp12
  E >> oscprob.comp13
  E >> oscprob.comp23
  unity          >> ws.sum.comp0
  oscprob.comp12 >> ws.sum.item12
  oscprob.comp13 >> ws.sum.item13
  oscprob.comp23 >> ws.sum.item23
```



$$P_{\mathrm{sur}} = P_0 + \sum_i \omega_i(\theta_{12}, \theta_{13}, \theta_{23}) \cos\left(C\frac{L\Delta m_i^2}{E}\right)$$

Figure 1. The neutrino oscillation probability formula, example of the binding in GNA and resulting computational graph

It may be noted that the example code is quite verbose and not suitable for the description of the large-scale models. For the general description the concept of *bundles* is developed. Bundles are used to facilitate the construction of small computational graphs and scale them based on a simple configuration. A bundle is a Python class that reads a dictionary containing the configuration, initializes a set of variables, instantiates and binds a set of transformations. The result of bundle execution is a small computational chain which may be then incorporated into the model.

To give a reference, a computational graph within GNA corresponding to the Daya Bay experiment contains about 500 nodes and 1000 edges in the minimal configuration, and depending on the structure of the model, it may contain around 2500 nodes. The latest version of the JUNO model contains around 2600 nodes and almost 5000 edges. To see the example computational graphs built within GNA, see [7].

The bundles handle only a creation of the partial graphs. Binding these parts is a distinct task handled via *GNA expressions*.

3. GNA expressions and the parser

GNA expressions are the concept that uses mathematical expressions to describe the relationships between transformations and bindings [1]. An expression is initialized with the following information: a) a mathematical expression, b) the definition of indices used in the expression, c) configurations for the bundles to be executed to provide expression elements. Each keyword in the expression is converted into a transformation or a variable. A keyword may carry indices which indicate multiple branches of computation. Expression is a valid Python code and they are parsed within a predefined Python environment using unsafe *exec* function. The current implementation does not support other (potential) backends and has no proper error reporting.

Therefore, it was decided to develop an independent GNA expression parser module, and implement the expressions as a *Domain Specific Language* (DSL) [8]. With this aim, we need to break the expression into *tokens* or *terminals*. Each token is a single atomic element of the language. In our case, these are transformations, variables, indices, namespaces, special predefined identificators (e.g. zero, unity), mathematical operations, etc. The process of converting a sequence of characters into a sequence of tokens is called *lexical analysis*, and a program that performs such analysis is called *lexical analyzer* (*lexer*). This analysis is usually implemented by using regular expressions. However, having a sequence of tokens from some input is not enough: we need to ensure that they form a valid expression in our language and they respect the syntactic structure (*syntax*, *grammar rules*) of the DSL. This phase is called *syntactic analysis* or *parsing*. The program or procedure that performs such analysis is called a *parser*. For a more detailed overview, we refer to [9].

For an explicit distinction between transformations and variables, it was decided to introduce the following rules: transformation (variable) is a sequence of 4 to 50 characters, the first of which is in upper (lower) case. Mathematical operations (+, -, *, /, @) can be performed on transformations and variables, however, it is impossible to add, subtract a transformation from a variable and vice versa (and it is also impossible to assign one to another). Matrix multiplication @ is only defined for transformations. Zero and unity transformation are reserved by the keywords *0, zero, null* and *1, unity, one*. Transformations, variables or operations may be labeled by *::label::* or kept in namespaces, which is written as *ns1.ns2.Transf*. Inline comments begin with a character # and are ignored along with spaces, tabs, and newlines.

In order to solve the graph scaling problem, *indices* were introduced. Using indices, it is possible to implement several variations of the same sub-graph without changing the code or structure of the original graph. Binding some graph with indices *i*, *j*, *k* will produce several graphs corresponding to all combinations of these indices. Transformations, variables or operations may be indexed explicitly using square brackets *[]* or implicitly by inheriting indices from child nodes. These indices may be reduced using operators with curly braces *{}*.

The parser was developed as part of a stand-alone module and based on the Lark package [10]. It supports fast and strict LALR(1) parsing algorithm, caching and grammar based on EBNF syntax. At the moment, the GNA grammar file has only 50-100 lines and fully describes the aforementioned tokens and rules, taking into account the corresponding priority of operations, and including some additional functionality. However, not to mention large-scale physical models, even for complex DSLs, it is necessary to very accurately formulate grammatical rules in order to eliminate ambiguities, therefore a set of unit tests is provided. The developed module also includes the graphs functionality, based on *PyGraphviz* [11] and full representativeness of any stage.

During the parsing of mathematical operations, for example, product or sum, the corresponding objects are automatically created, which are explicitly added to the namespace and computational graph. In order to make large auto-generated graphs readable without affecting either the code or their structure, the *pattern matching* was implemented. This procedure consists in setting a library of patterns (names, expressions and, if necessary, labels), using which the parser will replace the expressions encountered with the corresponding patterns.

Consider the following example: the parsing input

```
Transf1| Transf2+Functions[m]| variables[i]*(Matrix@Transf3*norm)
```

and the pattern matching library

```
NewTransf:
        expr: “Matrix@Transf3*norm”
        label: “Labeled transformation” .
```

The expression is parsed without actual knowledge of what data and functions are: each name is considered to be an index, label, namespace, operation, variable or transformation output. The call operation **|** means that the output, associated with the argument, is bound to the input, associated with a function. The transformation *Transf1* is applied to the sum of transformations *Transf2* and *Functions* with index *m*. The bundle that will provide *Functions* will provide an output for each variant of index *m*. Then, the *Functions[m]* is applied to the product of the variable *variables* with index *i* and the construction with matrix multiplication of *Matrix* and the product *Transf3*norm*.

The given graph contains indices *i* and *m*, so the resulting computational graph will consist of all variants of the original graph in accordance with the combinations of indices *i* and *m*. After parsing the expression, there is a pattern matching stage, at which the construction *Matrix@Transf3*norm* is replaced by the transformation *NewTransf*. The parsing tree is shown in [fig. 3]. Here the *sequence* node is the starting point for any parsing procedure. Explicit mathematical operations do not have their own tokens and are a special type of transformation. Any call operator **|** produces the child *arg* node of the corresponding transformation or operation.

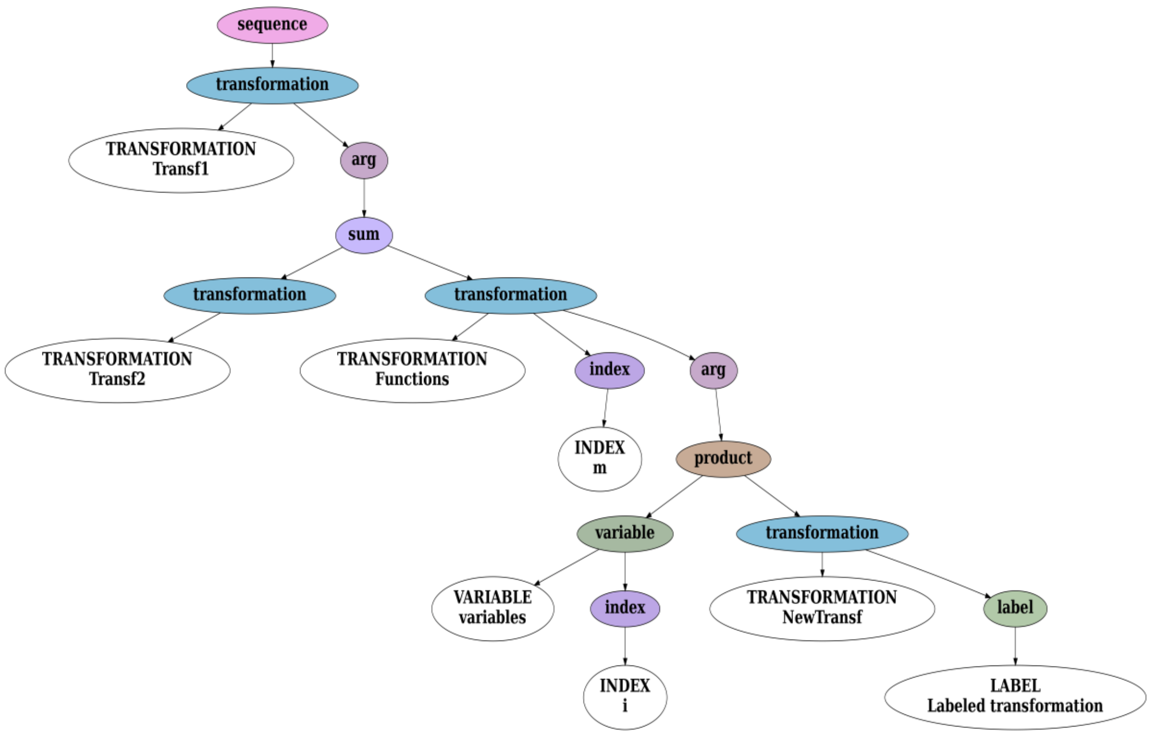


Figure 3. The parsing tree for the example above

At the next stage, a computational graph is built from the parsing tree, which stores all the information about the inputs and outputs. [Fig. 4] shows the computational graph corresponding to the considered example. Nodes like *production_0* or *sum_1* are automatically generated for explicit mathematical operations. Implicit indices are shown in parentheses.

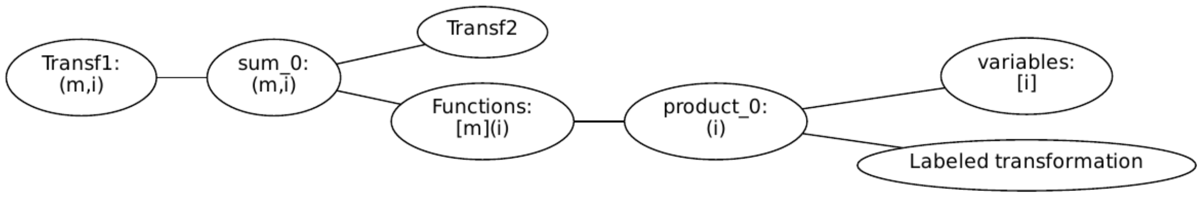


Figure 4. The resulting computational graph

4. Conclusion

The module, which implements the GNA DSL parser has been developed. It enables the user to describe the computational graph using a simple syntax. The graph may be annotated via pattern matching and also contains extra information describing branching (indices). The module is independent of other GNA parts and may be used to build the computational graphs for other *potential* backends. The syntax of the DSL is described by a concise grammar file and may be modified to provide alternative syntax depending on the requirements of the physical model. At the current stage, the developed module and the GNA core are being merged. The next step is to check and optimize the parser for the physical models used in the framework.

5. Acknowledgements

The research is supported by the Russian Science Foundation grant 21-42-00023. Konstantin Treskov is also supported by the grant 21-202-10 pf Association of Young Scientists and Specialists of JINR.

References

[1] Fatkina A., Gonchar M., Kalitkina A., Kolupaeva L., Naumov D., Selivanov D. and Treskov K. GNA: new framework for statistical data analysis // EPJ Web of Conferences, 2019, vol. 214, pp. 05024, ISSN 2100-014X, DOI: 10.1051/epjconf/201921405024 [arXiv:1903.05567 [cs.MS]]

[2] An F.P. et al. (Daya Bay Collaboration) // Phys. Rev. D 95, 2017, pp. 072006, DOI: 10.1103/PhysRevD.95.072006 [arXiv:1610.04802 [hep-ex]]

[3] Dennis J.B., Fosseen J.B., Linderman J.P. Data flow schemas // International Symposium on Theoretical Programming. Berlin, Heidelberg: Springer Berlin Heidelberg, 1974, pp. 187–216, ISBN 978-3-540-38012-2

[4] Abadi M. et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems // 2016, arXiv:1603.04467v2

[5] Paszke A. et al. Automatic differentiation in PyTorch. Available at: https://openreview.net/forum?id=BJJsrmfCZ (accessed 06.09.2021)

[6] Bezanson J., Edelman A., Karpinski S., Shah V.B. Julia: A Fresh Approach to Numerical Computing // SIAM Review, 59, 2017, pp. 65–98, doi: 10.1137/141000671

[7] Gallery of various graphs, generated based on GNA models. Available at: http://gna.pages.jinr.ru/gna/gallery.html (accessed 27.08.2021)

[8] Mernik M., Heering J., Sloane A.M. When and how to develop domain-specific languages // ACM Comput. Surv. 37, 4, 2005, pp. 316–344. DOI:https://doi.org/10.1145/1118890.1118892

[9] Aho A.V., Lam M.S.-L., Sethi R., Ullman J. D. Compilers: Principles, Techniques, and Tools. Boston, Massachusetts, USA: Addison-Wesley, 2006. ISBN 0-321-48681-1. OCLC 70775643

[10] The Lark package documentation. Available at: https://lark-parser.readthedocs.io (accessed 27.08.2021)

[11] The PyGraphviz web page. Available at: https://pygraphviz.github.io (accessed 27.08.2021)