EPJ Web of Conferences **251**, 03068 (2021) https://doi.org/10.1051/epjconf/202125103068
CHEP 2021

FuncADL: Functional Analysis Description Language

Mason Proffitt[1,*] and *Gordon* Watts[1,**]

[1]University of Washington

Abstract. The traditional approach in HEP analysis software is to loop over every event and every object via the ROOT framework. This method follows an imperative paradigm, in which the code is tied to the storage format and steps of execution. A more desirable strategy would be to implement a declarative language, such that the storage medium and execution are not included in the abstraction model. This will become increasingly important to managing the large dataset collected by the LHC and the HL-LHC. A new analysis description language (ADL) inspired by functional programming, FuncADL, was developed using Python as a host language. The expressiveness of this language was tested by implementing example analysis tasks designed to benchmark the functionality of ADLs. Many simple selections are expressible in a declarative way with FuncADL, which can be used as an interface to retrieve filtered data. Some limitations were identified, but the design of the language allows for future extensions to add missing features. FuncADL is part of a suite of analysis software tools being developed by the Institute for Research and Innovation in Software for High Energy Physics (IRIS-HEP). These tools will be available to develop highly scalable physics analyses for the LHC.

1 Introduction

Analyses of high energy physics (HEP) data typically consist of executing a carefully designed algorithm on millions of collision events. The algorithm is specific to the analysis but identical for all events in a given data-taking period. The per-event results are aggregated over the entire dataset, such as in the form of histograms or a table recording the number of events passing certain selections. One of the most common ways to write analysis code is essentially a for-loop over events, generally using C++ and the ROOT software framework [1, 2]. This procedure tightly couples the analysis code to both the file format of the data and the steps of program execution.

There have been several recent efforts to provide more declarative interfaces for data analysis in order to simplify analysis code and to allow taking advantage of optimization via parallelization and vectorized operations. Within the ROOT package, the `RDataFrame` class aims to provide these features by drawing on the concept of data frames present in other platforms like R, Apache Spark, and the Python package pandas [3]. Coffea is a CMS-focused project designed to handle cuts and histograms similarly to `RDataFrame`, but entirely in Python, utilizing the Uproot package for reading and writing ROOT files [4].

*e-mail: masonLp@uw.edu
**e-mail: gwatts@uw.edu

Many of the issues faced in HEP data analysis are similar to those mitigated by work in database management systems. Two important aspects of database management are data query languages and data independence [5]. Query languages such as SQL are naturally declarative, as they only specify relationships between data fields. Data independence is the concept that the interface for accessing data should not depend on how and where the data is actually stored. There is not yet a widely adopted declarative interface to write HEP analyses across a variety of underlying storage formats. The object of the FuncADL project is to provide this interface via a query language that maintains data independence.

The FuncADL project encompasses several different aspects: the query language, code generation, and implementations that tie these two steps together. The following sections address these components. A simple implementation is demonstrated with example analysis tasks drafted by the HEP software community, and there is a discussion of implications and future plans.

2 Query language

FuncADL has been designed as a query language within Python [6]. The fundamental base class is `EventDataset`, which generically represents a dataset of collision events, regardless of the storage format or location. All queries are constructed by applying operations on an instance of `EventDataset`. The query operations were inspired by LINQ (Language INtegrated Query), a feature built into the C# language [7]. Query operators always act on a sequence of objects, which may be the sequence of events or a sequence that exists within each event, such as a jet collection. The most important query operations can be categorized as projection, filtering, or aggregation operators.

Projection operators apply a transformation to each element of a sequence. This transformation is based on the properties of each element. For example, this can be used to select only the properties of an event relevant for a particular analysis and to disregard the rest of the information. `EventDataset` itself is effectively a trivial projection operator passing all event information through. There are two non-trivial projection operators: `Select` and `SelectMany`. Both of these take a lambda function as an argument that specifies the transformation to apply to each sequence element. The result of `Select` is simply the sequence of results of this transformation. The `SelectMany` projection operator expects this transformation to produce a sequence for each element, and the operator concatenates all of these sequences. This allows flattening collections across all events, such as all jets in a dataset.

Filtering operations filter out elements of a sequence based on a certain condition. The filtering operator is `Where`. `Where` expects a lambda function argument that results in a boolean value. An element of the input sequence only passes through the filter if applying the lambda function to it evaluates to `True`. Otherwise the sequence element is dropped.

Aggregation operators map an input sequence to a single value. These act on each element sequentially and reduce the dimensionality. The generic form is `Aggregate` operation, which takes a lambda function and a seed value as arguments. The lambda function is executed on the first element of the sequence with the seed value as a second argument. The resulting value becomes the seed value for applying the function to the second element, which provides the seed value for the third element, and so on. There are several common aggregation operations that are specializations of the `Aggregate` operator which are implemented directly in the language for convenience. These include the operators `Count`, `Max`, `Min`, and `Sum`. `Count` outputs the number of elements in the input sequence. `Max`, `Min`, and `Sum` expect the input sequence to be numerical values and output the maximum element, the minimum element, and the sum over all elements, respectively.

Only an example subset of operators have been enumerated here. There are other operations supported for more specialized usage. In addition, several native Python types and operations retain identical functionality within FuncADL. Lists, tuples, and dictionaries are allowed within queries. All arithmetical, boolean, and bitwise Python operators are supported as well. One important special operation supported in FuncADL is `Zip`. `Zip` takes several sequences of the same length and turns them into one sequence by bundling together the elements at a given position in each input sequence. The functionality is similar to the Python builtin class `zip` [8]. However, rather than only yielding tuples, FuncADL's `Zip` operator also allows zipping dictionaries together to output a sequence of dictionaries. This makes it possible to create new ad hoc data structures inside a query.

Useful queries are generally composed of several operations. These can be applied in succession on the same sequence, or they can be nested such that some of the query operations are run on sequences within each event. The level of nested queries can continue to any depth of object structure present in the data.

3 Code generation

The key to achieving data independence with a query language is having backends that can handle the details of how to actually execute a query. For FuncADL, this means taking a representation of the user's query and producing a script or executable that will apply the required transformations to the data. Currently two FuncADL backend libraries exist: one utilising the ATLAS AnalysisBase software [9] and one using Uproot [10]. These backends are designed to run on xAOD [11] files and flat ROOT ntuples, respectively. Here a "flat ntuple" refers to a ROOT `TTree` that does not require any external schema to interpret its data structures. For example, this includes the CMS NanoAOD [12] format and potentially the developing ATLAS DAOD_PHYSLITE [13] format, as well as most ntuples produced by physics analysis teams.

The FuncADL query is internally represented by an abstract syntax tree (AST). An AST is composed of nodes corresponding to syntactic elements of Python and FuncADL, such as a lambda function or a `Select` operation. Each node contains links to child nodes, corresponding to the components of that syntactic element (for example, the arguments of a lambda function and the body of the function). The AST is built using the `ast` module included in Python [14]. The backend implementations of FuncADL take the AST of a query and traverse the tree with an `ast.NodeTransformer` (or an `ast.NodeVisitor`). For each node, these `NodeTransformers` build a representation of the corresponding operation in the appropriate framework for the particular backend. This process translates a query AST into standalone code that will perform the retrieval and transformations specified by the query. The output is the generated code text, which can be compiled or executed as needed.

One of the currently available backend implementations is for the xAOD format. This is the format used by ATLAS for data files produced by the collaboration for general use by analysis teams. The xAOD format is a specially structured ROOT file produced by the Athena software framework. Because of the data structures used, it is generally necessary to run a collaboration-specific framework like AnalysisBase in order to usefully read and extract information from the files. Thus a FuncADL backend for xAOD files was written that generates C++ code utilizing the ATLAS xAOD libraries. This allows accessing xAOD data structures from within the FuncADL query language, such as leptons and jet collections.

Another backend available for FuncADL is based on Uproot and Awkward. Uproot is a Python package completely independent from any ROOT code that is able to read ROOT files. Awkward is a related Python package that provides an interface for accessing and manipulating arrays with potentially jagged dimensions [15]. These packages are designed

to be lightweight and fast, benefiting from compiled vectorized optimizations. Uproot is capable of reading any ROOT `TTree`s that contain standard data types like strings, integers, floating point values, and variable-length vectors. This makes it ideal for operating on ROOT files with a simple structure like NanoAOD (CMS) or DAOD_PHYSLITE (ATLAS). The Uproot FuncADL backend translates a query AST into generated Python source code for a function that can be evaluated on a data file and returns the selected and transformed values as an Awkward array.

New backends for FuncADL can be implemented at any time. All that is required is to write a new `NodeTransformer` that can take in a FuncADL query AST and produce a script or executable which applies the necessary transformations to a data file of the appropriate format. It is possible to have multiple backends that can operate on the same format. For example, a prototype for an `RDataFrame`-based backend was created for an early version of FuncADL. The Uproot and `RDataFrame` backends are both able to run on flat ntuples, so this provides the option for alternative backends and performance benchmark comparisons between them.

4 Implementations

The final piece of implementing FuncADL is tying the frontend query language to the backends via an interface that is able to execute a query and return the result to the user. There are a few different options for this, depending on where the query should actually be executed. The main options available are running via a suite of services called ServiceX [16] or running the full transformation locally in the same process that is submitting the query. The choice is determined by which subclass of `EventDataset` is used as the base for the FuncADL query. In both cases, the communication between the frontend parsing the user's query and the backend generating and executing code is done via a format called Qastle [17]. The query built by combining an `EventDataset` instance and operations acting on it is only evaluated on demand. That is, the query is only executed when the `.value()` method is called. Therefore the construction of the query can be done in multiple lines and can be separated from the evaluation step.

Query AST Language Expressions (Qastle) is a way of specifying data queries detached from any host language. Qastle is essentially a plain text format consisting of LISP-like [18] expressions corresponding to the nodes of a query AST. This format was created to provide a human-readable interface between the frontend and the backends. The Qastle format also removes extraneous information from the generic Python AST structure that is not relevant to FuncADL queries. The FuncADL frontend translates a user query into Qastle and passes it to the executor associated with the particular `EventDataset` subclass used.

ServiceX is a full-featured data delivery service. ServiceX itself consists of several services that perform dataset resolution, code generation, and data transformation. It is possible to run ServiceX locally, although it is intended to be run on a cluster as a highly scalable platform. Query execution via ServiceX is possible through the classes `ServiceXSourceXAOD` and `ServiceXSourceUpROOT` for the xAOD and Uproot backends, respectively. The FuncADL frontend package produces a Qastle-formatted query and sends this to ServiceX. The FuncADL backends drive the code generation and data transformation steps. The result is then sent back to the analysis user.

It is also possible to run FuncADL queries outside of ServiceX. For example, the `UprootDataset` class provides a way to execute queries using the FuncADL Uproot backend. With this method, the code generated for a query by the backend is simply run locally after calling the `.value()` function on the query. This functionality is useful for testing and

for applying transformations to small datasets that don't necessitate the use of cluster-scale resources.

The modular design of the frontend and backends of FuncADL allows for the possibility of adding further methods of execution in the future. It is possible to drop in a replacement for the FuncADL frontend as well. Any package that is capable of producing the Qastle format can be used in conjunction with the backends. This means that it is possible to run queries with the FuncADL backends that were not even written in the FuncADL query language. This strategy has been implemented by the package `tcut_to_qastle` [19], which translates ROOT `TCut`-formatted strings into Qastle, and these `TCut` queries can then be executed via ServiceX.

5 Query examples

In order to craft concrete examples of FuncADL queries, a list of eight ADL benchmark tasks were used [20]. This list was inspired by conversations within the HEP Software Foundation Data Analysis Working Group on typical simple data transformations needed in writing an analysis. The tasks are performed on CMS open data that has been converted to the NanoAOD format. For simplicity and because the example dataset was relatively small (16 GiB), these queries were run using local execution. They were run in Jupyter notebooks via CERN's SWAN service [21].

The following two setup lines are common to all of the benchmark task implementations:

```
from func_adl_uproot import UprootDataset
ds = UprootDataset('Run2012B_SingleMu.root')
```

The argument to `UprootDataset` is the path of the data file. The selections in the first four benchmark tasks have been fully implemented in FuncADL, as shown in Table 1. Only the FuncADL queries are shown here, but the resulting histograms made from the returned arrays can be seen in the GitHub repository [22]. The last four benchmark tasks could not be implemented directly in FuncADL because of limitations in the current version, as shown in Table 2. The missing features are a cross join (or cartesian product), sequence sorting, and the calculation of invariant mass. However, it is possible to apply partial selections with FuncADL and implement the remaining aspects of each benchmark task by using the FuncADL selection output in combination with other Python packages.

#	ADL benchmark task and corresponding FuncADL query
1	Plot the missing E_T of all events.
	`ds.Select(lambda event: event.MET_pt)`
2	Plot p_T of all jets in all events.
	`ds.SelectMany(lambda event: event.Jet_pt)`
3	Plot p_T of jets with $\|\eta\| < 1$.

	`ds.SelectMany(lambda event: Zip({'pT': event.Jet_pt,`
	`'eta': event.Jet_eta})\`
	`.Where(lambda jet: abs(jet.eta) < 1)\`
	`.Select(lambda jet: jet.pT))`

4	Plot the missing E_T of events that have at least two jets with $p_T > 40$ GeV.

	`ds.Where(lambda event: event.Jet_pt\`
	`.Where(lambda pT: pT > 40)\`
	`.Count() >= 2)\`
	`.Select(lambda event: event.MET_pt)`

Table 1. Implemented benchmark tasks. Only the FuncADL query is provided rather than the plot, which is a histogram of the elements in the result. These plots can be seen in the GitHub repository.

#	ADL benchmark task	Reason not implemented
5	Plot the missing E_T of events that have an opposite-sign muon pair with an invariant mass between 60 and 120 GeV.	Missing cross join and invariant mass
6	Plot p_T of the trijet system with the mass closest to 172.5 GeV in each event and plot the maximum b-tagging discriminant value among the jets in the triplet.	Missing cross join, sorting, and invariant mass
7	Plot the sum of p_T of jets with $p_T > 30$ GeV that are not within 0.4 in ΔR of any lepton with $p_T > 10$ GeV.	Missing cross join
8	For events with at least three leptons and a same-flavor opposite-sign lepton pair, find the same-flavor opposite-sign lepton pair with the mass closest to 91.2 GeV and plot the transverse mass of the missing energy and the leading other lepton.	Missing cross join, sorting, and invariant mass

Table 2. Unimplemented benchmark tasks. The features missing from FuncADL that are needed for full implementation of each task are indicated.

6 Discussion

As shown in the previous section, the data selections necessary for the first four ADL benchmark tasks were successfully implemented in FuncADL using the Uproot backend. This demonstrates how the query language can express simple projections of sequences of events and object collections within each event, filtering of these sequences, and aggregation across elements. These operations already encompass many basic data query use cases.

The selections required for the last four ADL benchmark tasks could not be expressed entirely within FuncADL, which indicates potential areas for improvement. For example, the ability to sort a sequence by key values is an important query language feature that is missing. Simple cross joins are in principle already well defined in the FuncADL syntax, but limitations in the Uproot backend do not yet support these. The last feature needed for full implementation of the ADL benchmark tasks is the calculation of invariant mass. The full details of the calculation could be written directly in the FuncADL query, but this would be inconvenient and error-prone. As this is a common domain-specific function for HEP analyses, a more satisfactory solution would be to natively support four-vector operations. All of these mentioned features are planned to be supported by future versions of FuncADL.

There are some limitations of FuncADL that are instilled by design. As a query language, the functionality is limited to data extraction and transformation. FuncADL is not intended to be used to run an entire analysis from beginning to end. Calculating and applying scale factors, systematic variations, and creating actual histograms from selected data are examples of analysis tasks that are not well suited to FuncADL. However, FuncADL is intended to be highly modular, such that it provides a uniform interface for querying data, either manually or via other analysis software. This can reduce the burden on other analysis software that can instead focus on driving high-level analysis tasks.

Maintaining this narrow scope and modularity for FuncADL allows it to remain flexible and generalize well. The xAOD backend is not preconfigured to use any particular collections or object properties, so it can be used with any xAOD file compatible with the same AnalysisBase release. Similarly, the Uproot backend is not tied to any particular format of `TTree`. This still allows for any package to be built on top of FuncADL that does impose a particular schema for an analysis user's convenience.

7 Conclusions

The FuncADL project provides a declarative query-based interface to specify, retrieve, and transform data needed for HEP analyses. This interface is not dependent on the underlying file format or data structures. The query interface is identical whether executing the selection locally or remotely, either on a single machine or a cluster. A subset of the query functionality has been demonstrated on the ADL benchmark tasks. Some limitations have been identified, but these can currently be compensated for by combined usage with other packages and can be implemented directly by future extensions to the language.

8 Acknowledgements

This work was supported by the National Science Foundation under Cooperative Agreement OAC-1836650.

References

[1] *TTree Class Reference*, https://root.cern/doc/master/classTTree.html (2021), accessed: 2021-02-28

[2] *AnaAlgorithm Introduction*, https://atlassoftwaredocs.web.cern.ch/ABtutorial/alg_basic_intro/ (2018), accessed: 2021-02-28

[3] D. Piparo, P. Canal, E. Guiraud, X. Pla, G. Ganis, G. Amadio, A. Naumann, E. Tejedor, EPJ Web of Conferences **214**, 06029 (2019)

[4] L. Gray, N. Smith, A. Novak, D. Thain, D. Taylor, P. Fackeldey, C. Carballo, P. Gessinger, J. Pata, D. Kondratyev et al., *Coffeateam/coffea: Version 0.7.1* (2021), https://doi.org/10.5281/zenodo.4552694

[5] S. Abiteboul, R. Hull, V. Vianu, *Foundations of Databases: The Logical Level*, 1st edn. (Addison-Wesley Longman Publishing Co., Inc., USA, 1995), ISBN 0201537710

[6] *func_adl*, https://github.com/iris-hep/func_adl (2020), accessed: 2021-02-28

[7] *Language Integrated Query (LINQ)*, https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/ (2017), accessed: 2021-02-28

[8] *Built-in Functions*, https://docs.python.org/3/library/functions.html (2021), accessed: 2021-02-28

[9] ATLAS Collaboration, *Athena* (2019), https://doi.org/10.5281/zenodo.2641997

[10] J. Pivarski, H. Schreiner, C. Burr, D. Davis, N. Hartmann, A. Novak, ChristopheR-appold, G. Stark, J. Bendavid, M. Peresano et al., *scikit-hep/uproot4: 4.0.4* (2021), https://doi.org/10.5281/zenodo.4543730

[11] A. Buckley, T. Eifert, M. Elsing, D. Gillberg, K. Koeneke, A. Krasznahorkay, E. Moyse, M. Nowak, S. Snyder, P. van Gemmeren, J. Phys. Conf. Ser. **664**, 072045 (2015)

[12] Rizzi, Andrea, Petrucciani, Giovanni, Peruzzi, Marco, EPJ Web Conf. **214**, 06021 (2019)

[13] Elmsheuser, Johannes, Anastopoulos, Christos, Boyd, Jamie, Catmore, James, Gray, Heather, Krasznahorkay, Attila, McFayden, Josh, Meyer, Christopher John, Sfyrla, Anna, Strandberg, Jonas et al., EPJ Web Conf. **245**, 06014 (2020)

[14] *ast — Abstract Syntax Trees*, https://docs.python.org/3/library/ast.html (2021), accessed: 2021-02-28

[15] J. Pivarski, P. Das, I. Osborne, A. Biswas, N. Smith, H. Schreiner, N. Hartmann, D. Kalinkin, C. Burr, L. Gray et al., *scikit-hep/awkward-1.0: 1.1.2* (2021), https://doi.org/10.5281/zenodo.4535217

[16] *ServiceX - Data Delivery for the HEP Community*, https://github.com/ssl-hep/ServiceX/ (2021), accessed: 2021-02-28

[17] *qastle*, https://github.com/iris-hep/qastle (2021), accessed: 2021-02-28

[18] *Common Lisp HyperSpec*, http://www.lispworks.com/documentation/lw50/CLHS/Front/index.htm (2005), accessed: 2021-02-28

[19] *TCutToQastleWrapper*, https://github.com/ssl-hep/TCutToQastleWrapper (2020), accessed: 2021-02-28

[20] *adl_benchmarks_index*, https://github.com/iris-hep/adl-benchmarks-index/ (2021), accessed: 2021-02-28

[21] *The Swan Service*, https://swan.web.cern.ch/swan/ (2021), accessed: 2021-02-28

[22] *func_adl_benchmarks*, https://github.com/masonproffitt/func_adl_benchmarks/blob/master/swan_notebook.ipynb (2021), accessed: 2021-02-28