Declarative interfaces for HEP data analysis: FuncADL and ADL/CutLang

C Huh[1], M Proffitt[2], H B Prosper[3], S Sekmen[1], B Sen[4], G Unel[5], and G Watts[2]

[1] Kyungpook National University
[2] University of Washington
[3] Florida State University
[4] Middle East Technical University
[5] University of California Irvine

Abstract. Analysis description languages are declarative interfaces for HEP data analysis that allow users to avoid writing event loops, simplify code, and enable performance improvements to be decoupled from analysis development. One example is FuncADL, inspired by functional programming and developed using Python as a host language. FuncADL borrows concepts from database query languages to isolate the interface from the underlying physical and logical schemas. The same query can be used to select data from different sources and formats and with different execution mechanisms. FuncADL is one of the tools being developed by IRIS-HEP for highly scalable physics analysis for the LHC and HL-LHC. FuncADL is demonstrated by implementing example analysis tasks designed by HSF and IRIS-HEP. Another language example is ADL, which expresses the physics content of an analysis in a standard and unambiguous way, independent of computing frameworks. In ADL, analyses are described in human-readable text files composed of blocks with a keyword-expression structure. Two infrastructures are available to render ADL executable: CutLang, a runtime interpreter written in C++; and adl2tnm, a transpiler converting ADL into C++ or Python code. ADL/CutLang are already used in several physics studies and educational projects, and are adapted for use with LHC Open Data.

1. Introduction

An analysis description language is a domain-specific language (DSL) that is sufficient to completely specify a collider physics analysis. Such a language presents an analysis in an unambiguous, declarative manner. Emphasis is placed on the physics content of the analysis and not algorithmic design of its execution. Details that are not essential to the analysis are abstracted away. This leads to code that is easier to write and review, and optimization of performance and resource utilization can be decoupled from the analysis description. In the following sections, we present two examples of analysis description languages: FuncADL and ADL/CutLang.

2. FuncADL

FuncADL (Functional Analysis Description Language) is an embedded DSL that has been implemented within Python [1, 2]. It incorporates features from both functional programming and database query languages. In FuncADL, a dataset of collision events is viewed as an object

that can be queried against in order to filter and transform data as desired. Much of the functionality of FuncADL was based on LINQ (Language INtegrated Query), a part of the C# language [3]. LINQ-like query operators act on sequences of objects, which could be the sequence of collision events or a sequence that exists within each event, such as a collection of jets.

Most of the query operations can be categorized as projection, filtering, or aggregation operators. Projection operators transform the elements of a sequence by applying the same function to each element. This can be used to pick out only the properties of an event that are relevant for a given analysis. Filtering operations drop certain elements of a sequence based on defined criteria. Aggregation operators map an entire sequence to a single value. Most of the basic native Python types and operators retain their functionality in FuncADL as well.

In general, queries are composed of several different 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 increase to any depth of object structure present in the data. FuncADL utilizes lazy evaluation, so that constructing queries does not execute any resource-intensive code until the results are explicitly desired.

As a query language, FuncADL is naturally limited to data extraction and transformation. It is not intended for FuncADL to be used to run an entire physics analysis. Calculating and applying scale factors, systematic variations, and creating 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 software that can instead focus on driving high-level analysis tasks.

When evaluated, a FuncADL query is interpreted by a backend implementation that is appropriate for the underlying data. The backend library translates the FuncADL operations into lower-level code that will actually execute the query. Multiple backend libraries exist for running queries on different data formats. This separation between the query interface and the execution decouples the FuncADL frontend from any particular framework or file format.

FuncADL queries are represented internally by an abstract syntax tree (AST). An AST is composed of nodes corresponding to semantic elements of Python and FuncADL, such as equality comparisons or query operators. Each node contains links to its child nodes, corresponding to the components of that semantic element (for example, the name of a query operator and its arguments). The backend converts each AST node into a native representation in the language or framework being used and then executes the generated code. The result is seamlessly passed back to the analysis user via the frontend interface.

Currently three FuncADL backend libraries have been fully developed: an xAOD backend, a CMS Run 1 backend, and an Uproot backend. The xAOD backend can be used to run queries on data in the xAOD format from the ATLAS experiment [4]. Similarly, the CMS Run 1 backend can run queries on data files that were produced by the CMS experiment during Run 1 of the LHC. These two backends both produce C++ code that is compiled against the appropriate experiment-specific libraries. The Uproot backend can operate on any flat ROOT ntuples and uses the Python packages Uproot and Awkward Array [5, 6]. Here a "flat ntuple" refers to a ROOT `TTree` that does not require any external schema to interpret its data structures [7].

Additional FuncADL backends can easily be created to extend its use to new data formats. This only requires writing a package that will translate a FuncADL query AST into a script or executable that can apply the described transformations to data in a file of the target format. There can also be multiple backends that operate on the same format.

Even with a particular backend, the are multiple ways to run a FuncADL query. One option available is to execute the queries locally. This still provides the benefits of the declarative interface of FuncADL for specifying a selection of data. A more practical option for running on large datasets is for the query to be submitted to a cluster. The most important use case

targeted by FuncADL so far is execution via ServiceX.

ServiceX is a flexible, high-performance data delivery service for high energy physics [8]. It is also possible to run ServiceX itself locally, although it is intended to be run on a cluster as a highly scalable platform. ServiceX consists of several services that perform dataset resolution, code generation, and data transformation. A frontend package allows constructing queries via the FuncADL interface that when executed will be sent to a ServiceX endpoint. ServiceX handles the data retrieval and starts parallel workers that run a FuncADL backend implementation. The results are then sent back to the analysis user. Communication between the frontend that parses a user's query and the backend that generates and executes code occurs through a language called Qastle [9]. Query AST Language Expressions (Qastle) is a method of specifying data queries that is not tied to any host language. Qastle is a plain text format that consists of LISP-like [10] expressions that correspond to nodes of a query AST.

FuncADL's modular design allows easily adding alternative models of execution. It is also possible to drop in a replacement for the FuncADL frontend. Any package that produces the Qastle format can be used together with the FuncADL backend implementations. This makes it possible to use the FuncADL backends to run queries that were not even originally written in the FuncADL query language. Such a strategy has already been implemented by the `tcut_to_qastle` package [11], which translates ROOT `TCut`-formatted strings to Qastle so that these `TCut` queries can be executed with ServiceX.

In order to produce concrete examples of FuncADL queries, a set of eight ADL benchmark tasks were used [12]. This list was inspired by discussions in the HEP Software Foundation Data Analysis Working Group on simple data transformations representative of those required in an analysis. The tasks were run on CMS open data with the Uproot backend. For simplicity and because of the relatively small size of the example dataset (16 GiB), the queries were run via local execution.

The full code and output for the benchmark task implementations can be found in a GitHub repository [13]. The selections for the first six tasks were fully implemented with FuncADL. The last two tasks could not readily be directly implemented with FuncADL due to functionality limitations of the current version. However, it is possible to apply intermediate selections via FuncADL and then implement the remaining portion of each benchmark task by using the FuncADL query output together with other Python packages. It is planned to extend future versions of FuncADL to make implementations of these last two tasks easier.

The following snippet is an example of the FuncADL query for the fourth benchmark task:

```
from func_adl_uproot import UprootDataset
UprootDataset('Run2012B_SingleMu.root')\
.Where(lambda event: event.Jet_pt.Where(lambda pT: pT > 40)\
                                  .Count() >= 2)\
.Select(lambda event: event.MET_pt)
```

The `UprootDataset` object represents the sequence of collision events in the data sample. The argument to `UprootDataset` is the path of the data file. The `Where` operator is used to filter events based on how many jets with $p_{\mathrm{T}} > 40$ GeV are present (via the `Where` and `Count` operators). Finally, the `Select` operator is used to extract only the missing transverse momentum (`MET_pt`) from the events passing the filter.

3. ADL/CutLang

Analysis Description Language (ADL) is a declarative language designed as an external DSL with its dedicated syntax. It aims to describe the physics content of a collider analysis in a standard and ambiguous way. In this approach, description of the analysis physics algorithm

is completely decoupled and independent from software frameworks. ADL can be parsed and processed by any tool or framework capable of recognizing its syntax.

The idea of a DSL for use by LHC physicists was thoroughly discussed first time at the Les Houches Physics at TeV Colliders Workshop in 2015, where a group of experimentalists and theorists agreed that a standardized way to describe LHC analyses could greatly benefit the HEP community. Motivations and design features of such a language were debated and an first prototype of a DSL called the "Les Houches Analysis Description Accord (LHADA)" was created [14]. LHADA included the main analysis operations such as physics object and event selection definitions. In parallel, another DSL called CutLang was being designed with a focus on exploring the capabilities of runtime interpretation. CutLang had started as a project to provide a practical way for beginner students with minimal experience in programming to perform complex analyses. Since both the LHADA and CutLang languages were based on the same design principles, and have similar syntax, in 2019 they were merged, under the name ADL, combining the best of both into a single language.

The domain scope of ADL focuses on event processing. It includes definitions of simple and composite objects (jets, electrons, Z bosons, etc.), event variables (hadronic transverse momentum H_T, aplanarity, etc.) and event selections ($n_{jets} > 2$, $H_T > 500$, etc.). Histogram operations are also included. Expression of systematic uncertainties is intended to be within the ADL scope, however the syntax for these is still under development. ADL at its current state can also host expressions of analysis results, such as counts and uncertainties.

A plain, easy-to-read text file called the ADL file hosts the analysis physics algorithm transcribed with the ADL syntax. The ADL file consists of multiple varieties of blocks separating object, variable and event selection definitions. Blocks are built based on a keyword-expression structure, where keywords specify analysis concepts and operations. Current ADL syntax includes mathematical and logical operations, comparison and optimization operators, reducers, 4-vector algebra and standard HEP functions (e.g. $\delta\phi$, δR). 1- and 2-dimensional fixed or variable bin histograms for object and event quantities can also be defined. Existing selection results including counts and uncertainties (e.g. published by experiments) can be documented in the ADL file. Further details on the ADL syntax can be found in [19]. Some analyses may contain variables with complex algorithms non-trivial to express with the ADL syntax (e.g. stransverse mass, aplanarity) or non-analytic variables (e.g. machine learning models). Such variables are encapsulated in self-contained, standalone external functions that can be referenced from within an ADL file. Figure 1 shows an excerpt from a simple analysis written with the ADL syntax.

ADL itself is only a description and requires accompanying tools to be rendered executable. The framework-independence signifies that analyses written in ADL can be processed with any tool or framework that can recognize and parse the ADL syntax. A typical ADL analysis workflow can be seen in Figure 2. The system takes as input the ADL file, external functions (when required) and events in ROOT format. The output includes cutflows, counts, histograms, and optionally, selected events. Two approaches have been studied so far. One is the transpiler approach, where an analysis written in ADL syntax is converted into a general purpose language. An example is the prototype `adl2tnm` transpiler, where a Python script translates ADL into C++ code that can be compiled into an executable program [15].

The other approach is that of runtime interpretation, where ADL can be directly executed, without the need of intermediate translation or compilation of the analysis description. This approach was explored via the CutLang runtime interpreter. [17, 18, 19]. CutLang is written in C++ and works in any modern Unix environment. It is based on ROOT classes for Lorentz vector operations and histogramming. CutLang performs automatic ADL parsing through the state-of-the-art tools Lex & Yacc. CutLang interpreter is extended with a framework to manage input/output operations. It can automatically recognize and process multiple input event

formats regularly used in HEP, such as Delphes fast simulation, CMS NanoAOD, ATLAS/CMS Open Data, LVL0, FCC, while more formats can be easily added. CutLang produces output in ROOT files, which contains cutflows, bins and histograms for each event selection region in a separate TDirectory, as well as the ADL description itself for provenance tracking. CutLang is available in multiple platforms including Docker and Conda. A Jupyter kernel is available for processing ADL syntax directly via CutLang through binder or Conda.

ADL and CutLang are already being used for physics studies. They are employed for implementing various physics analyses for experimental and phenomenological studies [20, 16]. A growing LHC ADL analysis database is available in GitHub [22], which contains both complete and simplified analyses that can be processed using CutLang. The database can be used as a physics information source, serve reinterpretation studies, analysis queries, comparisons and combinations. It is intended a first step towards long term analysis preservation. ADL and CutLang are also used in schools for training students in HEP analysis [21]. Moreover, work is in progress to establish ADL/CutLang as an accessible analysis model for ATLAS and CMS Open Data.

4. Conclusions

We presented two examples demonstrating the use of domain specific languages for high energy physics data analysis, which aim to describe an analysis in a completely unambiguous way. These

```
# OBJECTS
object goodJet
  take jet
  select pT(jet) > 30
  select abs(eta(jet)) < 3

object goodMuon
  take Muon
  select pT(Muon) > 20
  select abs(eta(Muon)) < 2.4

object goodEle
  take Ele
  select pT(Ele) > 20
  select abs(eta(Ele)) < 2.5

object goodLep
  take union(goodEle, goodMuo)
```

```
# EVENT VARIABLES
define HT = fHT(jets)
define MT1 = Sqrt( 2*pT(goodLep[0]) * MET*(1-cos(phi(METLV[0]) - phi(goodLep[0]) )))

# EVENT SELECTION
region SR
  select size(goodJet) >= 2
  select MET > 300
  select HT > 500
  select MET / HT <= 1.1
  select Size(goodLep == 0
  select dphi(METLV[0], goodJet[0]) > 0.5
  select dphi(METLV[0], goodJet[1]) > 0.3
  select size(goodJet) >= 3 ? dphi(METLV[0], goodJet[2]) > 0.3 : ALL
  select size(goodJet) >= 4 ? dphi(METLV[0], goodJet[3]) > 0.3 : ALL
  histo hgoodJet , "number of jets", 15, 0, 15, size(goodJet)
  histo hMETHT , "MET vs HT (GeV)", 20, 300, 1300, 40, 500, 2000, MET, HT
```

Figure 1. Description of a simple analysis selection (inspired by supersymmetry searches) with the ADL syntax. This ADL can be directly processed over events with CutLang.

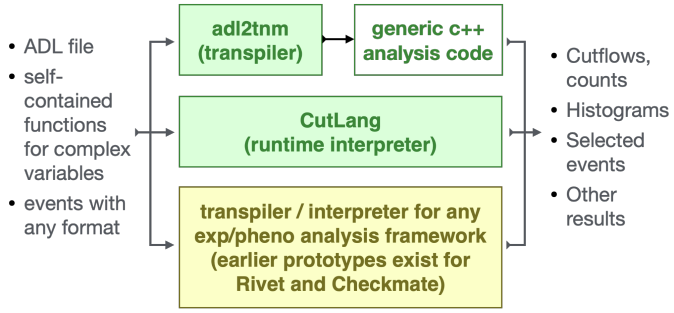

Figure 2. ADL analyses flow with different tools. Inputs to and outputs from a typical ADL analysis.

languages typically provide a declarative interface to specify the physics content of an analysis without the details of its execution. FuncADL is an embedded DSL within Python, inspired by functional programming and query languages. ADL is an external DSL with its own runtime interpreter (CutLang) and transpiler (adl2tnm). Such declarative interfaces make analyses easier to write, understand and communicate, along with simplifying their long-term analysis preservation. FuncADL and ADL show the feasibility of the declarative DSL approach, which will be particularly relevant as we move towards the high-luminosity LHC era with increasing data and ever more complex physics analyses.

Acknowledgments

The work of MP and GW is supported by the National Science Foundation under Cooperative Agreement OAC-1836650. The work of SS is supported by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education under contracts NRF-2021R1I1A3048138, NRF-2018R1A6A1A06024970 and NRF-2008-00460.

References

[1] func_adl, https://github.com/iris-hep/func_adl
[2] M. Proffitt and G. Watts, "FuncADL: Functional Analysis Description Language," EPJ Web Conf. 251 03068 (2021) doi:10.1051/epjconf/202125103068 [arXiv:2103.02432 [physics.data-an]]
[3] Language Integrated Query (LINQ), https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/
[4] ATLAS Collaboration, Athena, https://zenodo.org/record/4772550
[5] J. Pivarski, *et. al.* scikit-hep/uproot4, https://doi.org/10.5281/zenodo.6081474
[6] J. Pivarski, *et. al.* scikit-hep/awkward-1.0, https://doi.org/10.5281/zenodo.5750763
[7] TTree Class Reference, https://root.cern/doc/master/classTTree.html
[8] ServiceX, https://github.com/ssl-hep/ServiceX
[9] qastle, https://github.com/iris-hep/qastle
[10] Common Lisp HyperSpec, http://www.lispworks.com/documentation/lw50/CLHS/Front/index.htm
[11] TCutToQastleWrapper, https://github.com/ssl-hep/TCutToQastleWrapper
[12] adl_benchmarks_index, https://github.com/iris-hep/adl-benchmarks-index
[13] func-adl-demo, https://github.com/masonproffitt/func-adl-demo/blob/master/demo.ipynb
[14] G. Brooijmans, *et. al.* "Les Houches 2015: Physics at TeV colliders - new physics working group report," [arXiv:1605.02684 [hep-ph]].
[15] G. Brooijmans, *et. al.* "Les Houches 2017: Physics at TeV Colliders New Physics Working Group Report," [arXiv:1803.10379 [hep-ph]].
[16] G. Brooijmans, *et. al.* "Les Houches 2019 Physics at TeV Colliders: New Physics Working Group Report," [arXiv:2002.12220 [hep-ph]].
[17] S. Sekmen and G. Ünel, "CutLang: A Particle Physics Analysis Description Language and Runtime Interpreter," Comput. Phys. Commun. **233** (2018), 215-236 doi:10.1016/j.cpc.2018.06.023 [arXiv:1801.05727 [hep-ph]].
[18] G. Unel, S. Sekmen and A. M. Toon, "CutLang: a cut-based HEP analysis description language and runtime interpreter," J. Phys. Conf. Ser. **1525** (2020) no.1, 012025 doi:10.1088/1742-6596/1525/1/012025 [arXiv:1909.10621 [hep-ph]].
[19] G. Unel, S. Sekmen, A. M. Toon, B. Gokturk, B. Orgen, A. Paul, N. Ravel and J. Setpal, "CutLang V2: towards a unified Analysis Description Language," doi:10.3389/fdata.2021.659986 [arXiv:2101.09031 [hep-ph]].
[20] A. Paul, S. Sekmen and G. Unel, Eur. Phys. J. C **81** (2021) no.3, 214 doi:10.1140/epjc/s10052-021-08982-4 [arXiv:2006.10149 [hep-ph]].
[21] A. Adıgüzel, O. Çakır, Ü. Kaya, V. E. Özcan, S. Öztürk, S. Sekmen, I. Turk Cakir and G. Ünel, "CutLang as an Analysis Description Language for Introducing Students to Analyses in Particle Physics," Eur. J. Phys. **42** (2021), 035802 doi:10.1088/1361-6404/abdf67 [arXiv:2008.12034 [hep-ph]].
[22] ADL LHC Analyses Database, https://github.com/ADL4HEP/ADLLHCanalyses