Agile Construction of Data Science DSLs (Tool Demo)

Conference Paper · October 2019

DOI: 10.1145/3357765.3359516

4 authors, including:

Artur Andrzejak
Universität Heidelberg
114 PUBLICATIONS **2,086** CITATIONS
SEE PROFILE

Diego Elias Costa
Concordia University Montreal
20 PUBLICATIONS **66** CITATIONS
SEE PROFILE

Agile Construction of Data Science DSLs (Tool Demo)

Artur Andrzejak
Heidelberg University
Heidelberg, Germany
artur@uni-hd.de

Kevin Kiefer
Heidelberg University
Heidelberg, Germany
kiefer@stud.uni-heidelberg.de

Diego Elias Costa
Heidelberg University
Heidelberg, Germany
costa@informatik.uni-heidelberg.de

Oliver Wenz
Heidelberg University
Heidelberg, Germany
o.wenz@stud.uni-heidelberg.de

Abstract

Domain Specific Languages (DSLs) have proven useful in the domain of data science, as witnessed by the popularity of SQL. However, implementing and maintaining a DSL incurs a significant effort which limits their utility in context of fast-changing data science frameworks and libraries.

We propose an approach and a Python-based library/tool NLDSL which simplifies and streamlines implementation of DSLs modeling pipelines of operations. In particular, syntax description and operation implementation are bundled together as annotated and terse Python functions, which simplifies extending and maintaining a DSL. To support ad hoc DSL elements, NLDSL offers a mechanism to define DSL-level functions as first-class DSL elements.

Our tool automatically supports each DSL by code completions and in-editor documentation in a multitude of IDEs implementing the Microsoft's Language Server Protocol. To circumvent the problem of a limited expressiveness of a external DSL, our tool allows embedding DSL statements in the source code comments of a general purpose language and to translate the DSL to such a language during editing.

We demonstrate and evaluate our approach and tool by implementing a DSL for data tables which is translated to either Pandas or to PySpark code. A preliminary evaluation shows that this DSL can be defined in a concise and maintainable way, and that it can cover a majority of processing steps of popular Spark/Pandas tutorials.

CCS Concepts • **Human-centered computing → User interface programming**; *Natural language interfaces*; • **Information systems → Information integration**.

GPCE '19, October 21–22, 2019, Athens, Greece

Keywords DSL development, Code generation, Assisted editing and IntelliSense, Data analysis frameworks, Apache Spark, Python Pandas

ACM Reference Format:
Artur Andrzejak, Kevin Kiefer, Diego Elias Costa, and Oliver Wenz. 2019. Agile Construction of Data Science DSLs (Tool Demo). In *Proceedings of the 18th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '19), October 21–22, 2019, Athens, Greece.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3357765.3359516

1 Introduction

Domain Specific Languages (DSLs) have proven useful in developing software systems, and are increasingly adopted by practitioners in a multitude of application domains [9, 13, 24]. They greatly facilitate communication between domain experts and developers, and can make software development more efficient. For example, DSLs offer a more problem-oriented, terse and (partially) declarative description of the software solution. They can also hide irrelevant implementation details, and eliminate "syntactic noise".

On the other hand, DSLs impose several challenges. Implementing a DSL and a supporting editing environment might incur significant development costs and require specialized knowledge of parser/compiler technologies and corresponding tools. Since DSLs are limited in scope, they might not be able to express all elements of a particular solution. This might require integration and mixing with general-purpose languages like in the case of SQL and e.g. C/C++/Java (this is typically the case for so-called external DSLs [13]). Further disadvantages include need for developers to learn yet another language, and possibly more complex build, testing, and debugging processes.

We attempt to address these issues for a family of constrained DSLs, namely languages which model *chains* or *pipelines* of operations. Software solutions which use the concept of a pipeline of operations are frequently encountered in modern data processing and analysis libraries, for example in Apache Spark, in libraries for data science, like Pandas

and sklearn for Python, or the R project (e.g. dplyr package[1] with the pipeline operator '%>%'). More general usage of the pipeline of operations can be found in Java Streams API and a variety of dataflow languages and tools [21]. Consequently, pipeline-oriented DSLs have potentially a wide range of applications.

We propose an approach and a tool to develop and support usage of such DSL families. By constraining the language design spectrum we are able to provide some benefits to DSL developers and DSL "end-users". The DSL developers can define and implement individual DSL operations (components of a pipeline) in a terse and maintainable code, without need for knowledge of complex API or sophisticated parsing and code generation technologies. In particular, the syntax of DSL operations and their key properties are stated in a declarative way, as annotations and Python doc strings of the code-generating Python function. This meta information is leveraged by our tool for e.g. simplified argument parsing, and automated support for code recommendations. In addition, we provide a mechanism to declare and expand DSL-level functions. This can be used to define new DSL elements in an ad-hoc way.

Our tool allows generating target code from DSL in a compiler-like mode, or in an interactive mode, i.e. during editing of a target language script. For the later case our tool offers DSL code completions and embedded DSL documentation for a large number of IDEs which support the Microsoft's Language Server Protocol [8].

As a proof-of-concept we implemented a DSL for processing and analysis of data tables. Our implementation can generate target code for Python/Pandas or Apache PySpark. The DSL code is embedded in Python comments and so DSL and regular Python code can be mixed together (in the interactive tool usage). This property partially circumvents the problem of limited DSL breadth. Our DSL allows a spectrum of queries similar to a relational algebra. A preliminary evaluation shows that this DSL can express a majority of typical data processing steps for the above-mentioned frameworks.

This paper has the following structure. Section 2 describes the tool architecture, the approach and library for supporting development of pipeline-oriented DSLs, and the characteristics of the data table DSL. Section 3 contains a preliminary evaluation of the tool and the data table DSL. Section 4 describes related work and Section 5 contains conclusions.

Tool availability and GPCE Demo Outline. The code editor part of NLDSL i.e. *NLDSL.edit* (with *NLDSL.lib* and the data table DSL) is available as an online IDE (Theia) at http://129.206.61.41:3000/. The GPCE Demo Outline is available as a web page at https://bit.ly/GPCEdemoNLDSL.

[1]https://www.rdocumentation.org/packages/dplyr/

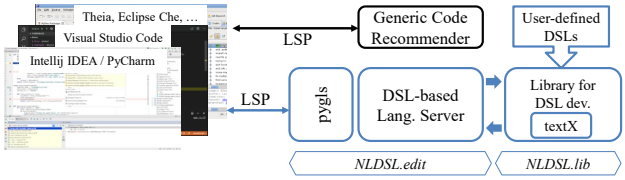

Figure 1. Architecture of our tool NLDSL.

2 Approach and Implementation

We describe here the tool architecture (Section 2.1), the key characteristics of the DSL families supported by our tool (Section 2.2, Section 2.3), and a proof-of-concept DSL for data table processing for Spark/Pandas (Section 2.4).

2.1 Tool Architecture

Our tool NLDSL consists of (i) a library *NLDSL.lib* for accelerated implementation of pipeline-oriented DSLs, and (ii) an environment *NLDSL.edit* supporting DSL editing and in-editor code generation for all IDEs supporting the Language Server Protocol (LSP, [8]). *NLDSL.lib* is implemented in Python and uses textX [9] for low-level DSL parsing. We describe this tool component and how to use it for DSL implementation in depth in Section 2.2 and Section 2.3. *NLDSL.lib* can be used alone (without *NLDSL.edit*). To generate target code in this mode we provide a compiler as a part of the library, which takes as input DSL file(s) and outputs generated (Python) code. A demonstration of using *NLDSL.edit* alone is included in the demo description.

NLDSL.edit uses *NLDSL.lib* and pygls [2], a generic Language Server "skeleton", to provide code completions for editors and IDEs supporting the LSP. In the current configuration the DSL-code completions work for Python files (extension .py). Since our DSL statements start with character '#', they are treated as comments by Python interpreters. In this way, users can mix DSL code and Python code in a single file. Figure 1 gives an overview of our tool with both components.

2.2 DSLs Structure

The DSLs supported by our tool consist of two types of statements: *evaluation* statements, which are translated to executable code, and *definition* statements, which are essentially ad-hoc definitions of functions within a DSL. The latter type of statement is explained in more depth in Section 2.3.3.

Figure 2 shows the general syntax of an evaluation statement as a railroad diagram. A statement is commenced by a DSL prefix '##' which allows to integrate DSL code into regular Python code as pseudo-comments (other prefixes like '//' for Java are possible). An optional assignment is followed by an expansion ("call") of an internal DSL function ('DSL_Function') (specified by a definition statement), or by a chain (or a pipeline) of DSL operations. The operations in a chain are separated by the pipe sign '|' and

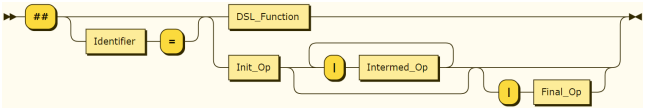

Figure 2. Grammar of the evaluation statement.

comprise a mandatory initialization operations ('Init_Op'), zero or more intermediate operations ('Intermed_Op'), and an optional final operation ('Final_Op'). Each of these operation types can be freely defined and implemented by a specific DSL. Initialization operations specify an object or data subject to processing. For convenience, we provide initialization operation 'on <identifier>' which determines an object/variable used as input for the pipeline.

The following listing shows some examples of valid grammar statements for the DSLs described in Section 2.4.

```
## x = on df | drop duplicates | groupy by df.data
    apply sum
## read 'myfile.csv' as csv | show
## x = 5 + 7 % 2
```

2.3 Defining and Implementing DSLs

Essentially, a DSL implementation consists of a set of Python functions corresponding to the desired pipeline operations (i.e. '*_Op' in Figure 2). Such *external rules* (described in detail in Section 2.3.1), are the backbone of the DSL and specify the translation into a target language/framework. They must be registered in a new class (inherited from 'CodeGenerator') which later serves as a code generator.

However, the DSL operations can be added and changed only by the DSL developers. To support DSL end-users in extending the DSL, we introduce DSL-internal functions which can be defined ad-hoc within a DSL script. Such functions summarize a chain of pipeline operations as a new DSL element which behaves similar to an operation and allows parameters (see Section 2.3.3).

2.3.1 Implementing External Rules (Operations)

As noted above, in a first step a DSL developer declares a code generator class 'GenCls', say (derived from 'CodeGenerator'). For each desired operation she must implement (and then register with 'GenCls' or its instance) a Python function with a following signature:

```
my_rule(code :str, args :list[str], env :dict[Any])
```

The first (mandatory) parameter is a string with already generated code, the second a list of DSL-parameters and the last is a dictionary containing environment variables (e.g. the name under which a certain module has been imported).

The registration of such a function with 'gen' can by done on the class level (static member), or on the instance level:

```
GenCls.register_function(my_rule, "rule name") # cls
    level
code_gen = GenCls()
code_gen["rule name"] = my_rule # obj level
```

DSL developers can use a dedicated decorator and a doc string with a special format to utilize several useful features:

1. Converting of DSL-parameters 'args' into a dictionary mapping variable names to their values.
2. Providing a list of expected next valid tokens, used for code recommendations and debugging.
3. Support for automated parsing of Boolean/comparison/arithmetic expressions among the DSL-parameters.
4. Support for automated name inference when while registering functions at code generator.

The format of such an enhanced function declaration is:

```
@grammar(doc :str, expr :ExpressionRule)
def my_rule(code :str, args :List[str], env
    :Dict[str:Any]):
"""Grammar:
     <name> (<var> | <var list> | <expr> |
         <keyword>)*
   Type:
     <Operation_Type>
"""
```

The section "Grammar" describes the DSL-syntax of the operation and its parameters. It can also contain additional lines specifying a set of possible values for selected parameters. The "Type" section indicates whether it is an initialization, an intermediate, or a final operation (see Section 2.2). This meta-information (sections "Grammar" and "Type") can be provided in a string 'doc' (the 1st decorator parameter) instead in the regular doc string. The optional parameter 'expr' specifies a class for parameter parsing (see Section 2.3.2).

For example, the following code defines and implements a DSL-operation 'group by' which emits Python/Pandas code. The parameters '$columns[$col]' is a list of column names, and parameter '$aggregation' takes a finite set of values:

```
gb_doc = """Grammar:
   group by $columns[$col] apply $aggregation
   aggregation := { min, max, sum, avg, mean, count }
Type:
   Operation
"""

@grammar(gb_doc)
def group_by(code, args):
   cols = list_to_string(args["columns"])
   return code + ".groupby({}).{}()".format(cols,
       args["aggregation"])
...
PandasCodeGenerator.register_function(group_by)
```

Exemplary usages of this DSL operations are:

```
## x = on df | group by df.col1 apply min
## x = on df | group by df.col1, df.col2 apply mean
```

2.3.2 Customized Expression Parsing

Our tool library provides extended support for parsing the DSL-parameters. In addition, it is possible to customize the parsing of expression by deriving from the 'ExpressionRule' class and passing the derived class to the grammar decorator. The derived class needs to specify how to map DSL-operators to strings in the target language. Moreover, one can also set whether an operator uses a postfix or an infix notation. The following listing shows an example of such a customization:

```
class MyExpressionRule(ExpressionRule):
def __init__(self, expr_name, next_keyword):
   super().__init__(expr_name, next_keyword)
   self.operator_map["and"] = " & "
   self.operator_map["+"] = " plus "
   self.operator_map["in"] = ".isin"
   self.operator_type["in"] =
        OperatorType.UNARY_FUNCTION
```

2.3.3 Extending DSLs via Internal Functions

A programmer using a DSL can easily define chains of DSL operations resembling parametrized functions on DSL-level. Such definitions are part of the DSL script and used during code generation. A definition consists of a declaration followed by '=' and a chain of existing DSL operations or DSL-functions on the right-hand-side. The grammar for the left-hand side is given by (EBNF notation):

```
LHS ::= "#$" name (keyword | var | expr | varlist)*
varlist ::= "$" identifier* "[" (keyword | var )+ "]"
var ::= "$" identifier
expr ::= "!" identifier
```

The following listing shows a definition of an internal rule 'only pos' and its usage in a DSL from Section 2.4:

```
#$ only pos $col = drop duplicates | select rows
     $col > 1
## N = on df | only pos df.colA | count
```

The generated codes for Pandas and Spark are, respectively:

```
N = df.drop_duplicates()[df.colA > 1].shape[0]
N = df.dropDuplicates().filter(df.colA > 1).count()
```

2.4 An Exemplary DSL for Data Tables

As a proof-of-concept for the utility of our tool we have implemented a DSL for table analysis and processing under Python. Our DSL can generate code for Pandas (a popular package for series and dataframe processing in Python) or for PySpark, i.e. Python bindings of Apache Spark (a framework for processing of massive data sets). When using editor support (Component B in Section 2.1), the code generation target can be specified via directives within Python comments (and even mixed in the same file) like '## target_code = spark'.

The utility of such a DSL is largely determined by the design of the DSL, in particular, the power of the DSL (or its"expressiveness") in terms of common operations for the target frameworks. To ensure a high level of DSL coverage, we have analyzed several popular "cheat-sheets" for Pandas and PySpark, as well as some tutorials for these frameworks. Based on this analysis we designed a DSL which covers most of the elements found in these sources, see Table 1.

Note that we do not attempt to cover all of the functionality in our DSL. Instead, we assume that developers will implement more specific functions directly in Python.

Our DSL attempts to be easy-to-understand (or, in the best case, even self-explanatory) yet concise. Thanks to code completion, long keywords are acceptable, and so we preferred better readability in the DSL design than compact but ambiguous keywords. The hurdle of learning and understanding the DSL is further reduced by explanations of the commands provided in the list of suggestions, if editor support (Component B) is used.

Table 1 shows the essential parts of our DSL. It covers functionality e.g. for data I/O, selection, aggregation, joins, set operations, various data transformations, and data inspection. In particular, the set of possible queries is similar to those available in Codd's relational algebra model [7].

3 Preliminary Evaluation of the Tool

A proper evaluation of the usefulness of our approach would require a well-designed user study with a sufficient number of participants. Due to time constrains, we refrain from performing such analysis and focus instead on assessing the effort of using the NLDSL to implement the two exemplary for data table DSLs Section 3.1, covering the operations defined in Table 1. Given a real-usage scenario we further evaluate what fraction of the analysis and data processing steps are covered by our exemplary DSLs Section 3.2.

3.1 Evaluation of NLDSL.lib

The exemplary data table DSL is implemented in 247 and 266 lines of Python-code, for Pandas and PySpark code generation respectively. As we implement the same set of operations in both Pandas and PySpark DSLs, they can share the documentation used by NLDSL to define the new grammar rules, specify types of arguments and illustrate the DSL usage with examples. This documentation comprises 412 lines, used by both DSLs. For instance, the operation load from is implemented in both DSLs as stated in Listing 1.

3.2 Expressiveness of Data Table DSL

Due to the popularity and demand for data analysis in current scientific landscape, tutorials for typical data analysis task are very common in the web. For our evaluation we use a popular DataCamp tutorial "Apache Spark Tutorial: ML with PySpark"[2]. This tutorial has multiple topics, but we focus only on the two topics within the scope targeted by our DSL: data exploration and data pre-processing. The relevant part of the tutorial contains 16 *processing steps*, i.e. smallest part of the code which can execute on its own.

To evaluate our DSL in context of Spark, we use unchanged source code from the tutorial. For Pandas, we manually translate each processing step into Pandas code, which yields only 14 processing steps, as some steps (such as creating RDDs to populate dataframes) are specific to PySpark. We do not use (another) tutorial directly for Pandas as it was impossible to find a tutorial with similar purpose/scenario as to the one for Spark. Moreover, many introductory tutorials for Pandas explain functions related to selecting individual values (e.g. `.loc` and `.iloc`), and functions related to the (row) index data structure of Pandas. Such functions make Pandas code fundamentally difficult to translate to other programming paradigms or frameworks (e.g. SQL, Spark), and typically make the code non-scalable. In our scenario, we assume that a developer considers the scenario of massive data sets and will avoid such functions right from the onset.

In our evaluation we attempt to express a basic processing step with our DSL. For each such step, we estimate whether it can be expressed by our current DSL design, or if our DSL cannot express this step at all. Overall, our DSL fully covers 12 out of 14 (85.8%) of Pandas processing steps and 14 out of 16 (87.5%) of PySpark processing steps. Four processing steps (two in Pandas and two in PySpark) could not be expressed or translated by our DSL, such as casting a column to a particular Python type (e.g., `.withColumn(..., cast(FloatType()))`), not currently supported by our DSLs.

[2]https://www.datacamp.com/community/tutorials/apache-spark-tutorial-machine-learning

Table 1. Overview of the DSL for data tables. Keywords are in **bold**, and choices from a list are underlined.

Category	DSL examples (prefix ## is omitted)
Table creation	**create dataframe from** *D* **with header** 'a', 'b'
I/O ops	result = **load from** 'some_path.json' **as** json
	on df | **save to** 'some_path.csv' **as** csv
Selection	result = **on** df | **select columns** df.a, 'b', 'c'
	... **select rows** df.s == 'A' **or** df.c1 > (14 + z)
	... **select rows** df.c1 > 0 **and** df.c2 **in** [3, 5, 7]
	... **select rows** df.c1 % df.c2 != 0
Aggregation	result = **on** df | **group by** df.c1 **apply** avg
	... **group by** df.c1, df.c2 **apply** count
Joins	**on** df | **join** inner df2 **on** 'c1', 'c2'
	... **join** left df2 **on** 'c1'
Set ops	**on** df | **intersection** df2
	... **difference** df2
	... **union** df2
Transform.	result = **on** df | **append column** df.c1*5 **as** 'c5'
	... **on** df | **drop duplicates**
	... **drop columns** df.x, 'y'
	... **replace values** old_value **by** new_value
	... **sort by** df.c1 descending, 'c2' ascending
	... **rename columns** c1 **to** c2, d1 **to** d2
	... **change type of** df.c1 **to** string
Inspection	**on** df | **show**
	... **show schema**
	... **count**
	... **describe**
	... **head** 20
Options	**target_code** = spark
	target_code = pandas

Listing 1. Definition of the load from external rule in NLDSL for Pandas and PySpark code translation.

```
LOAD_FROM_DOC = """Load a DataFrame from a file.

Examples:
   1. x = load from "my_file.json" as json
   2. x = load from "my_file.csv" as csv

Grammar:
   load from $path as $type
   type := { json, csv }

Args:
   path (variable): A string containing the path to
        a file.
   type (variable): The type of the file.

Type:
   Initialization
"""

# Pandas extension (from PandasCodeGeneration)
@grammar(docs.LOAD_FROM_DOC)
def load_from(code, args, env):
   read_fun = ".read_csv(" if args["type"] == "csv" \
     else ".read_json("
   return code + env["import_name"] + read_fun +
        args["path"] + ")"

# Spark extension (from SparkCodeGeneration)
@grammar(docs.LOAD_FROM_DOC)
def load_from(code, args, env):
   return "{}.read.format('{}').load({})".format(
     env["spark_name"], args["type"], args["path"])
```

4 Related Work

Domains relevant to our work are *Domain Specific Languages*, *accelerated scripting and coding*, and *low-code data analysis*.

Domain Specific Languages (DSLs) [13], [23], [9], have proven useful in a multitude of medium to large-scale projects by introducing highly readable and concise code with support for higher-level operations. While the underlying theories and scientific interest are still modest [23], [12], [16], DSLs are becoming increasingly popular in industry (for example, the industrial-grade database management system SAP HANA uses internally over 200 DSLs). Concepts related to DSLs are extensible languages like Racket [12], or enhancing libraries by "syntactic sugar" as in SugarJ [10].

A particular flavor of DSLs are internal or embedded DSLs which can seamlessly inter-operate with the underlying (typically general-purpose) language. However, internal DSLs offer only limited range of syntax and are typically not supported by IDEs. Contrary to this, external DSLs admit almost any syntax. Modern DSL engineering frameworks (Xtext [5], textX [9], Spoofax [23]) significantly lower the cost of developing such DSLs. Language workbenches (e.g. MPS [6], Xtext [5]) complement and extend such frameworks by providing editors with syntax checking and code completion for created languages, and by facilitating DSL parsing and code generation. This can greatly increase the acceptance of new DSLs (see [11] for a detailed comparison).

The disadvantages of external DSLs are the difficulty of interaction with (general-purpose) languages, and problems if the DSL capabilities are not sufficient. In our approach we generate code for a general-purpose language from an external DSL during the editing process, which largely eliminates the interoperability barrier. We also implemented a special Language Server to provide coding assistance to both our DSL and the "embedding" general-purpose language.

Accelerating scripting and coding (and as a special case, end-user development/end-user programming) comprises a multitude of approaches from software engineering. The most visible progress in this category stems from novel programming languages, introduction of software processes such as Scrum, proliferation of software testing, and advances in development tools (including Intelligent Development Environments, or syntax/error checkers). Nevertheless, the impact of the individual measures on programmers' productivity is hard to measure. Other noteworthy approaches include program synthesis (discussed above), visual programming via dataflow languages [21], block programming languages [3], and DSLs [13, 24], or more generally Language-Oriented Programming [12].

In the context of data analysis, dataflow languages [17, 21] have gained some popularity via tools such as [1] or KNIME [4]. Such approaches can greatly accelerate creation of small data processing pipelines but do not scale for larger projects.

Low-code data analysis. Multiple research fields tackle the challenge of making the process of data analysis and transformation more user-friendly and accelerating scripting and automation of processing. The essential directions are: visual analytics [25], mixed-initiative systems [19], [26], facilitating user involvement[27], learning data transformations by examples, [30], [20], [29], and data wrangling in various flavors [28], [22], [31].

Data wrangling (or data munging) refers to the process of interactive data transformations on structured or unstructured data. The most mature tool in this domain is Wrangler [22] by Trifacta which is based on the concept of Predictive Interaction [18]. Another popular tool is OpenRefine [32] (originally GoogleRefine) which allows batch processing of tabular data by menu selection and a DSL named GREL.

Learning data transformations by examples [30], [20], [29] is a special case of program synthesis. Such approaches (while still immature) offer a promise to greatly facilitate complex data analysis, especially for users with no or little programming skills. Several applications to data science exist [15]: extracting relations from spreadsheets, data transformations [20], and synthesizing regular expressions. So far the most widespread application is the FlashFill approach [14] as a component of Excel 2013+ and as *ConvertFrom-String* cmdlet in PowerShell. We consider program synthesis as a possible extension of our work.

5 Conclusions and future work

We proposed an approach and a tool for accelerated development and editing of pipeline-oriented DSLs, which are in particular suitable for data science scripting. Our approach allows declaring and implementing individual DSL operations in a compact and easy fashion. In particular, DSL syntax and certain properties are declared in Python doc strings.In addition, we provide a mechanism to declare and expand DSL-level functions which can be used to define new DSL elements in an ad-hoc way. Our tool allows generating target code from DSL in a "batch" mode (compiler-like), or during editing of a target language script. For the later case we provide DSL code completion based on the LSP protocol.

As a proof-of-concept we implemented a DSL for data tables which can generate code in Python/Pandas, Apache PySpark, or a mixture of them. This DSL has similar "expressibility" as a relational algebra. A preliminary evaluation shows that this DSL can cover a majority of typical data processing steps for the above-mentioned frameworks.

Our future work will include user studies with interviews in order to verify our hypotheses how programmers develop DSLs. We will also implement more code generation targets for our DSL, including Python/sklearn, deep learning frameworks like TensorFlow and PyTorch, as well as other languages like R (with dplyr/tidyR) and Matlab.

References

[1] 2018. Top 21 Self Service Data Preparation Software - Compare Reviews, Features, Pricing in 2019. (May 2018). https://www.predictiveanalyticstoday.com/data-preparation-tools-and-platforms/

[2] 2019. pygls, a Pythonic Generic Language Server. (2019). https://pypi.org/project/pygls/

[3] David Bau, Jeff Gray, Caitlin Kelleher, Josh Sheldon, and Franklyn Turbak. 2017. Learnable Programming: Blocks and Beyond. *Commun. ACM* 60, 6 (May 2017), 72–80. https://doi.org/10.1145/3015455

[4] Michael R. Berthold, Nicolas Cebron, Fabian Dill, Thomas R. Gabriel, Tobias Kötter, Thorsten Meinl, Peter Ohl, Christoph Sieb, Kilian Thiel, and Bernd Wiswedel. 2008. KNIME: The Konstanz Information Miner. In *Data Analysis, Machine Learning and Applications*. Springer, Berlin, Heidelberg, 319–326. https://doi.org/10.1007/978-3-540-78246-9_38

[5] Lorenzo Bettini. 2016. *Implementing Domain Specific Languages with Xtext and Xtend - Second Edition* (2nd ed.). Packt Publishing.

[6] Fabien Campagne. 2016. *The MPS Language Workbench Volume I: The Meta Programming System (Volume 1)* (3rd ed.). CreateSpace Independent Publishing Platform, USA.

[7] E. F. Codd. 1970. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM* 13, 6 (June 1970), 377–387. https://doi.org/10.1145/362384.362685

[8] Microsoft Corp. 2019. Language Server Protocol Specification. (2019). https://microsoft.github.io/language-server-protocol/specification

[9] I. Dejanović, R. Vaderna, G. Milosavljević, and Ž. Vuković. 2017. TextX: A Python tool for Domain-Specific Languages implementation. *Knowledge-Based Systems* 115 (Jan. 2017), 1–4. https://doi.org/10.1016/j.knosys.2016.10.023

[10] Sebastian Erdweg, Tillmann Rendel, Christian KÃďstner, and Klaus Ostermann. 2011. SugarJ: Library-based Syntactic Language Extensibility. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '11)*. ACM, New York, NY, USA, 391–406. https://doi.org/10.1145/2048066.2048099 event-place: Portland, Oregon, USA.

[11] Sebastian Erdweg, Tijs van der Storm, Markus VÃűlter, Laurence Tratt, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, GabriÃńl Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. 2015. Evaluating and comparing language workbenches: Existing results and benchmarks for the future. *Computer Languages, Systems & Structures* 44 (Dec. 2015), 24–47. http://www.sciencedirect.com/science/article/pii/S1477842415000573

[12] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, Sam Tobin-Hochstadt, and Marc Herbstritt. 2015. *The Racket Manifesto*. Technical Report. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik GmbH, Wadern/Saarbruecken, Germany. – pages. https://doi.org/10.4230/LIPIcs.SNAPL.2015.113

[13] Martin Fowler. 2010. *Domain Specific Languages* (1st ed.). Addison-Wesley Professional. 00681.

[14] Sumit Gulwani. 2015. Automating Repetitive Tasks for the Masses. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. ACM, New York, NY, USA, 1–2. https://doi.org/10.1145/2676726.2682621

[15] Sumit Gulwani. 2016. Programming by Examples (and its Applications in Data Wrangling). In *Verification and Synthesis of Correct and Secure Systems*. IOS Press. https://www.microsoft.com/en-us/research/publication/programming-examples-applications-data-wrangling/

[16] Gopal Gupta. 2015. Language-based Software Engineering. *Sci. Comput. Program.* 97, P1 (Jan. 2015), 37–40. https://doi.org/10.1016/j.scico.2014.02.010

[17] Philipp GÃűtze, Hoffmann Wieland, and Kai-Uwe Sattler. [n. d.]. Rewriting and Code Generation for Dataflow Programs. In *GI-Workshop on Foundations of Databases* (24.05.2016). NÃűrten-Hardenberg, Germany.

[18] Jeffrey Heer, Joseph Hellerstein, and Sean Kandel. 2015. Predictive Interaction for Data Transformation. In *Conference on Innovative Data Systems Research (CIDR)*. http://idl.cs.washington.edu/papers/predictive-interaction

[19] Eric Horvitz. 1999. Principles of Mixed-initiative User Interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '99)*. ACM, New York, NY, USA, 159–166. https://doi.org/10.1145/302979.303030

[20] Zhongjun Jin, Michael R. Anderson, Michael Cafarella, and H. V. Jagadish. 2017. Foofah: Transforming Data By Example. ACM Press, 683–698. https://doi.org/10.1145/3035918.3064034

[21] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. 2004. Advances in Dataflow Programming Languages. *ACM Comput. Surv.* 36, 1 (March 2004), 1–34. https://doi.org/10.1145/1013208.1013209

[22] Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. 2011. Wrangler: Interactive Visual Specification of Data Transformation Scripts. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '11)*. ACM, New York, NY, USA, 3363–3372. https://doi.org/10.1145/1978942.1979444

[23] Lennart C.L. Kats and Eelco Visser. 2010. The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '10)*. ACM, New York, NY, USA, 444–463. https://doi.org/10.1145/1869459.1869497 event-place: Reno/Tahoe, Nevada, USA.

[24] Tomaž Kosar, Sudev Bohra, and Marjan Mernik. 2016. Domain-Specific Languages: A Systematic Mapping Study. *Information and Software Technology* 71 (March 2016), 77–91. https://doi.org/10.1016/j.infsof.2015.11.001

[25] Joseph MacInnes, Stephanie Santosa, and William Wright. 2010. Visual Classification: Expert Knowledge Guides Machine Learning. *IEEE Comput. Graph. Appl.* 30, 1 (Jan. 2010), 8–14. https://doi.org/10.1109/MCG.2010.18

[26] Stephen Makonin, Daniel McVeigh, Wolfgang Stuerzlinger, Khoa Tran, and Fred Popowich. 2016. Mixed-Initiative for Big Data: The Intersection of Human + Visual Analytics + Prediction. IEEE, 1427–1436. https://doi.org/10.1109/HICSS.2016.181

[27] Protiva Rahman, Courtney Hebert, and Arnab Nandi. 2018. ICARUS: Minimizing Human Effort in Iterative Data Completion. *Proc. VLDB Endow.* 11, 13 (Sept. 2018), 2263–2276. https://doi.org/10.14778/3275366.3284970

[28] Vijayshankar Raman and Joseph M. Hellerstein. 2001. Potter's Wheel: An Interactive Data Cleaning System. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB '01)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 381–390. http://dl.acm.org/citation.cfm?id=645927.672045 00411.

[29] Mohammad Raza and Sumit Gulwani. 2017. Automated Data Extraction Using Predictive Program Synthesis. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA.*, Satinder P. Singh and Shaul Markovitch (Eds.). AAAI Press, 882–890. http://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/15034

[30] Calvin Smith and Aws Albarghouthi. 2016. MapReduce Program Synthesis. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, NY, USA, 326–340. https://doi.org/10.1145/2908080.2908102

[31] Michael Stonebraker, Ihab F Ilyas, Stan Zdonik, George Beskales, and Alexander Pagan. 2013. Data Curation at Scale: The Data Tamer System. *6th Biennial Conference on Innovative Data Systems Research* (2013).

[32] Ruben Verborgh and Max De Wilde. 2013. *Using OpenRefine* (1st new edition edition ed.). Packt Publishing. http://openrefine.org/