Johannes Luong, Dirk Habich, Wolfgang Lehner

AL: Unified Analytics in Domain Specific Terms

Erstveröffentlichung in / First published in:

DBPL 2017: The 16th International Symposium on Database Programming Languages, München 01.09.2017. ACM Digital Library, Art. Nr. 7.

DOI: https://doi.org/10.1145/3122831.3122835

Final edited form was published in "DBPL 2017: The 16th International Symposium on Database Programming Languages. München 2017",
Art. Nr. 7 , ISBN 978-1-4503-5354-0
https://doi.org/10.1145/3122831.3122835

AL: Unified Analytics in Domain Specific Terms

Johannes Luong, Dirk Habich, and Wolfgang Lehner
Technische Universität Dresden
Dresden, Germany
firstname.lastname@tu-dresden.de

ABSTRACT

Data driven organizations gather information on various aspects of their endeavours and analyze that information to gain valuable insights or to increase automatization. Today, these organizations can choose from a wealth of specialized analytical libraries and platforms to meet their functional and non-functional requirements. Indeed, many common application scenarios involve the combination of multiple such libraries and platforms in order to provide a holistic perspective. Due to the scattered landscape of specialized analytical tools, this integration can result in complex and hard to evolve applications. In addition, the necessary movement of data between tools and formats can introduce a serious performance penalty. In this article we present a unified programming environment for analytical applications. The environment includes *AL*, a programming language that combines concepts of various common analytical domains. Further, the environment also includes a flexible compilation system that uses a language-, domain-, and platform independent program intermediate representation to separate high level application logic and physical organisation. We provide a detailed introduction of *AL*, establish our program intermediate representation as a generally useful abstraction, and give a detailed explanation of the translation of *AL* programs into workloads for our experimental shared-memory processing engine.

CCS CONCEPTS

• **Information systems → Query languages for non-relational engines**; **Data analytics**; *Database query processing*; *Main memory engines*; *Online analytical processing engines*; *Computing platforms*;

ACM Reference format:
Johannes Luong, Dirk Habich, and Wolfgang Lehner. 2017. AL: Unified Analytics in Domain Specific Terms. In *Proceedings of DBPL 2017, Munich, Germany, September 1, 2017,* 9 pages.
https://doi.org/10.1145/3122831.3122835

1 INTRODUCTION

The potential for useful insights, predictions, and increased automatization, have motivated organizations to collect *big data* on all aspects of their respective endeavours. Due to a large variety of data sources and analytical goals, *big data* applications often deal with a variety of data types and processing algorithms. Just as Stonebreaker and Cetintemel have predicted [12], these applications depend on specialized engines and libraries to deal with particular aspects of data processing, such as relational data management, scientific computing, or graph analyses.

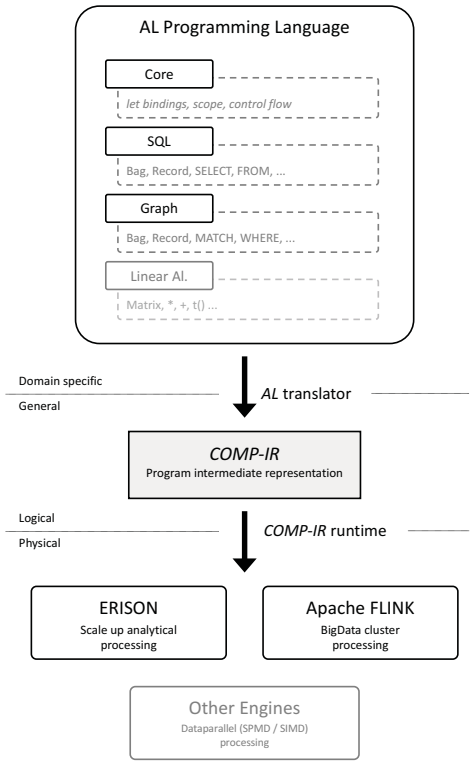

Figure 1: AL programming environment

Unfortunately, the ad-hoc integration of a variety of tools and systems comes with its own set of problems. Several authors criticize the introduction of tool specific programming models [4, 6, 7]. The combination of multiple such models leads to an explosion of complexity and each specialized programming model creates a lock-in effect that hinders adoption of future technology. Further, Alexandrov et al. [5] show that some of these models do not permit an application oriented programming style and require a quite detailed understanding of system internals in order to achieve good performance. This is in contrast to traditional data oriented languages such as SQL that isolate users from low-level details and facilitate automatic program optimization. The performance of using multiple tools and systems has been criticized as well. Palkar

Final edited form was published in "DBPL 2017: The 16th International Symposium on Database Programming Languages. München 2017",
Art. Nr. 7 , ISBN 978-1-4503-5354-0
https://doi.org/10.1145/3122831.3122835

et al. [11] show that bulk data movements between independent tools and libraries can degrade performance considerably.

In this paper, we introduce our unified data oriented programming model and show how this model deals with the aforementioned problems by introducing a novel layer of abstraction between high level programming languages and physical implementations. The discussion will revolve around three primary subjects: (1) the *AL* data processing language that integrates domain specific concepts of multiple application domains, (2) the domain-, language-, and platform independent program intermediate representation *COMP-IR*, and (3) an extensible translation system that maps *AL* programs to their *COMP-IR* representation and subsequently to an executable program of a physical execution engine.

Figure 1 provides a coarse overview of these components and their interactions. In the upper part, *AL* is shown as a container language that integrates several domain specific sublanguages such as a SQL dialect, a query language for graphs, and possibly others in future work. These languages function as the domain- and application oriented user interface of the programming environment. The intermediate program representation *COMP-IR*, in the center of the figure, provides a common compilation target for all domain specific sublanguages of *AL*. *COMP-IR* is a small, general language that is well suited to be translated into efficient programs for parallel processing engines. At the bottom of Figure 1, multiple processing engines are available to execute *COMP-IR* programs. These engines have to be extended with an adequate *COMP-IR* runtime, which translates the intermediate representation into physical instructions.

AL is a special purpose programming language for the definition of complex data processing programs. It integrates elements of various domain specific languages into a coherent multi-domain programming language. In this article, we discuss relational processing with a SQL derivate and graph processing with a sublanguage that resembles Cypher [1]. However, *AL* is not limited to these domains and we plan to add a linear algebra sublanguage in future work. Where possible, *AL*'s syntax resembles that of its ancestors, but in general, it has been chosen to be general and easily extensible. In particular, *AL* uses function and method calling syntax and it offers variable bindings to enable flexible composition of statements.

The *COMP-IR* intermediate representation is the central abstraction of the programming model. It provides a language, domain- and platform independent program representation that can express a large set of typical data processing applications. At its core, *COMP-IR* is an encoding of certain types of monad comprehensions [13], an elegant notation of programs that iterate over data collections to compute some result. Monad comprehensions have long been established as an adequate representation for certain query languages [8, 10]. But, due to their flexible data model and their general notion of computation, we find monad comprehensions to be especially useful as a unifying common basis for a multi domain data processing system.

To make practical use of *AL* and *COMP-IR*, we need a translator that can transform these representations into an executable program. As it is common in many compilers, we split the translation into two strictly separate phases. First, the *AL* frontend transforms an input program into an equivalent *COMP-IR* representation. Second, a runtime specific backend translates the *COMP-IR* representation into an executable program. The two compilation phases are implemented in separate applications that communicate exclusively via a JSON serialization of the *COMP-IR* program.

The main contributions of this paper are the following

(1) We introduce the programming language *AL* that currently supports relational- and graph pattern queries.
(2) We present the *COMP-IR* program intermediate representation and give a detailed explanation of the *COMP-IR* translator frontend for *AL*.
(3) We discuss an *COMP-IR* backend that generates executable programs for our experimental shared-memory data processing engine *Erison*.

The remainder of the article is structured as follows: in Section 2 we provide an in depth introduction of the *AL* programming language. In Section 3 we discuss the *COMP-IR* program intermediate representation and the *AL* translator frontend that compiles *AL* into *COMP-IR*. In Section 4 we take a look at *COMP-IR* runtimes and especially discuss the experimental *Erison* runtime for low latency, shared-memory processing. In Section 5 we take a look at related work, and in the final section we summarize our findings and consider directions for future work.

2 THE AL PROGRAMMING LANGUAGE

The primary goal of *AL* is to provide a unified query language for a variety of data processing domains. In contrast to typical query languages, *AL* uses function- and method calling syntax and enables query composition using variable bindings. Listing 1 shows an *AL* program that combines relational- and graph processing. The first part of the query consists of a simple relational query that computes the sum of all purchases of the current month for the active user. In the second part of the program, a graph query finds related products that could be advertised to the active user.

AL's sublanguage for relational queries closely resembles traditional SQL. The *FROM* clause designates *purchases_month* as source table of the query, the *WHERE* clause filters *purchases_month* based on its *user_id* column, and the *SELECT* clause creates a *sum* aggregation on the *price* column. The query computes a scalar of type double which is bound to the *currentBill* variable. *currentBill* is later used in the filter expression of the *WHERE* clause in the graph query. The use of the bound value is serves as example for the composition of queries in *AL*.

The graph query language resembles *Cypher* [1], which uses *patterns* to extract subgraphs of an overall graph. Patterns are specified as graphs themselves, using a textual representation of vertices and edges. In a nutshell, a pattern simply defines the shape of the subgraph that should be extracted by the query. Besides shape, the pattern can also specify predicates on vertex attributes or edge labels, to further constrain the set of matching subgraphs. The graph query in Listing 1 combines three patterns to find products that have been bought by users who also bought a product that the current user has previously looked at. The first pattern establishes the connection between the current user's vertex *u* and a product *p* that *u* has looked at. The second pattern adds the constraint that there has to exist an additional user *u2* who has bought *p* at some

Final edited form was published in "DBPL 2017: The 16th International Symposium on Database Programming Languages. München 2017", Art. Nr. 7 , ISBN 978-1-4503-5354-0
https://doi.org/10.1145/3122831.3122835


```
def recommendetProducts(uid: Long): ALPromise =
  AL"""
    currentBill =
      SELECT(
        sum(price)
      ).FROM(
        purchases_month
      ).WHERE(
        purchases_month.user_id == $uid
      )

    relatedProducts =
      MATCH(
        u[User](id == $uid) -visited-> p[Product],
        u2[User] -bought-> p,
        u2 -bought-> p2[Product]
      ).WHERE(
        ABS(u.avg_m_bill - u2.avg_m_bill) < 100.0,
        p2.price < u.avg_monthly_bill - currentBill
      ).RETURN(
        p2.id AS product_id,
        p2.url AS product_url
      )

    return relatedProducts
  """
```

Listing 1: Graph and relational query

point. Finally, the last pattern adds the requirement that there also has to be another product *p2* that has been bought by *u2*. Following the path patterns, the *WHERE* clause further constrains the set of matching subgraphs. In particular, the difference between the average monthly bills of *u* and *u2* can not exceed 100.0 and the price of *p2* hast to be lower than the difference of the average monthly bill and the current bill. The graph query is bound to the name *relatedProducts* which is eventually returned as the result of the *AL* program.

AL uses function- and method calling syntax as well as types to emulate the clauses of query languages. The *SELECT* function, for example, returns an object of type *SelectExp* that defines a *FROM* method. *FROM*, on the other hand, returns an object of type *BagExp* with methods such as *WHERE*, *GROUP_BY* and so on. Every *AL* program has to return a single result object and only the queries that are necessary to compute that object will be executed by the engine. However, this rule does not reduce the expressiveness of the language as *AL* functions are side effect free. In addition, *AL* programs can only return objects of certain types. Types such as *SelectExp* or *MatchExp* are considered to be internal because their objects do not represent a complete query. The set of types that *can* be returned by programs make up the data model of *AL*. Currently, the following types are supported:

Scalar	IntExp, DoubleExp, StringExp
Structured	RecordExp
Composite	BagExp

Many query languages involve the use of symbols, whose meaning is defined by an implicit context. In Listing 1, the meaning of *sum*(*price*) depends on a definition of *price*, which is provided by the subsequent FROM clause. *AL* treats any expression that contains implicit names as a lambda expression that abstracts over these names. Therefore, *sum*(*price*) is equivalent to $\lambda\, price.\, sum(price)$. This rule enables name binding of all elements of typical query languages and clearly defines the semantics of composing expressions with implicit names.

The first line of the listing is not part of the *AL* program, but belongs to a larger Scala[1] application that embeds the *AL* program as part of its logic. *AL* is implemented as a Scala library and *AL* programs can easily be embedded into Scala programs using the `AL"..."` string interpolator. The interpolator's primary use is the import of Scala values into *AL* programs. This can be seen in the relational query where *$uid* is used to access the *uid* parameter of the surrounding *recommendetProducts* Scala function. Internally, the interpolator simply concatenates *AL* code and Scala values into a single program string and forwards that string to a compiler function. That function translates the program into *COMP-IR* form, sends it to a *COMP-IR* runtime, and returns a result handle to the Scala application. As an alternative to the interpolator, the compiler function can also be used directly, which makes it easy to create standalone or web based *AL* clients.

3 *COMP-IR* AND THE *AL* TRANSLATOR

AL integrates a diverse set of query languages into a common framework. As can be seen in Figure 1, *COMP-IR* is the central element of the programming environment that enables this integration. *COMP-IR* accomplishes this by providing a common language that is both general enough to represent important query languages, but also specific enough so that important optimizations for data intensive applications can be expressed in it. The *AL* translator is an extensible compilation framework that transforms *AL* program strings into *COMP-IR* documents. The compilation process involves a sequence of program representations and transformation passes which we will discuss in detail in Section 3.1. We plan to extend *AL* with additional domain specific languages in future work and therefore put an emphasis on making the translator extensible. In Section 3.2, we show how the translator framework uses inheritance to add new language elements and transformation passes in a completely composable manner.

3.1 Representations of an *AL* program

The translation process involves four different program representations, as is depicted in Figure 2. Except the *AL* program itself, each representation is derived from the previous one and can be subject to a sequence of transformation passes. At first, the translator uses Scala library functions to parse the *AL* program string into a Scala abstract syntax tree (AST). The AST is a detailed, tree shaped, structural representation of the *AL* program. For our purposes, the AST has several drawbacks. On the one hand, it is designed for the feature rich general purpose programming language Scala. Therefore, simple domain specific concepts are spread over extensive AST subtrees and even simple domain specific transformations have to define complex tree matching and construction logic. On the other hand, the AST is a difficult target for control and data flow analyses, primarily because of the complex representation of jump target- and name binding positions in a graph.

[1] http://scala-lang.org/

Final edited form was published in "DBPL 2017: The 16th International Symposium on Database Programming Languages. München 2017", Art. Nr. 7 , ISBN 978-1-4503-5354-0
https://doi.org/10.1145/3122831.3122835

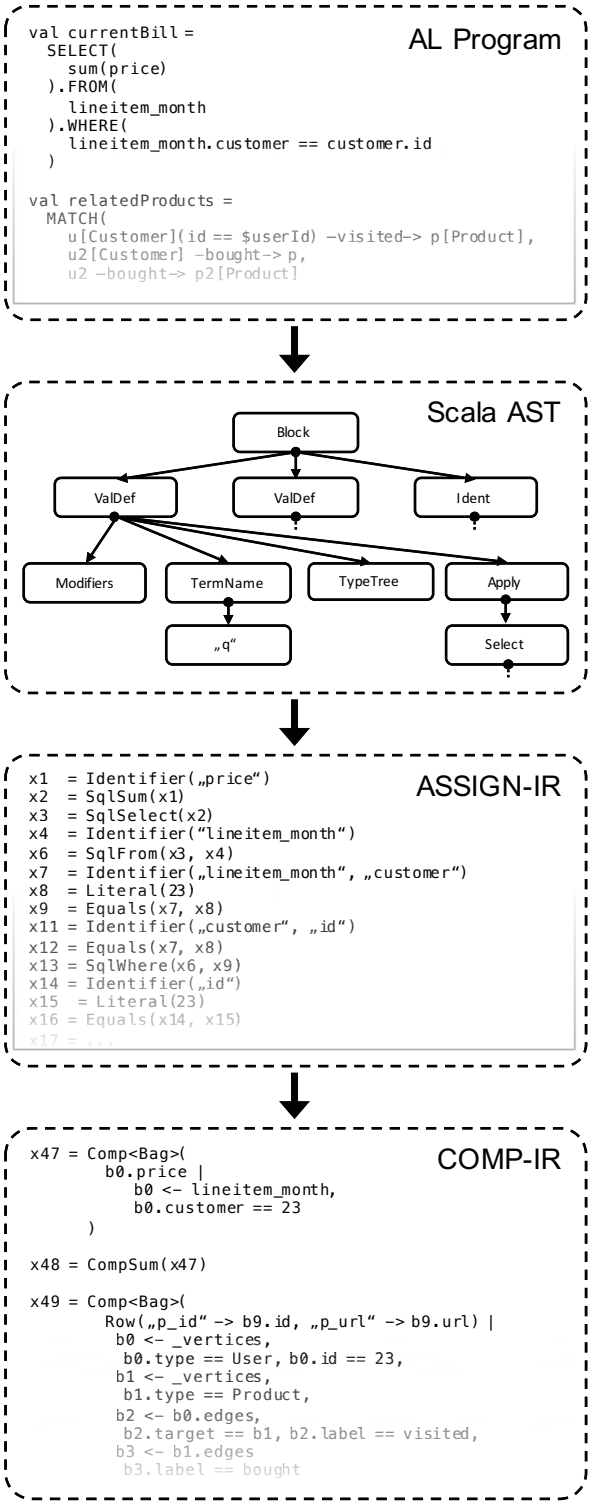

Figure 2: Representations of an AL program

Due to these limitations, the translator transforms the AST into the more convenient *ASSIGN-IR* representation. Conceptually, an *ASSIGN-IR* program is a directed graph of *blocks* and each block is a sequence of variable assignments. A variable is assigned to exactly once and each value is the result of a combinator expression over primitive values and previously assigned variables. The combinators represent the semantic vocabulary of *AL* and its domain specific languages. Some of the combinators are of general use, such as *Identifier*, *Literal*, or *Equals*. Others, such as *SqlSelect* or *SqlFrom*, represent domain specific concepts. The vocabulary of *ASSIGN-IR* can be easily extended by adding additional domain specific combinators. We use *ASSIGN-IR* as a generic code representation that can easily express more specialized concepts via *well known* combinator names. This is especially useful in combination with Scala's pattern matching abilities, which allow to identify combinators by name and parameters. In summary, *ASSIGN-IR* is a generic, easily extensible, and easy to deal with code representation, that explicitly represents control- and data flow.

COMP-IR is an *ASSIGN-IR* extension that adds combinators for monad comprehensions and a small set of comprehension specific functions. Monad comprehensions provide an elegant and domain independent language for the selection, composition and filtering of collections of data. Monad comprehensions can express the relational algebra but are not limited to relational processing. For example, they can also be applied to ordered sequential data structures such as arrays. *ASSIGN-IR* is transformed into *COMP-IR* by mapping domain specific combinators such as *SqlSelect(...)* onto comprehension combinators such as *Comp<Bag>(...)*. The *COMP-IR* transformation is finished, when all domain specific combinators have been replaced with comprehensions combinators and the program is in pure comprehension form. In the next sections we are going to discuss monad comprehensions in more detail and define abstract translation rules for several *AL* language elements.

Monad comprehensions. Monad comprehensions are closely related to the set builder notation of mathematics and list comprehensions of programming languages such as Haskell or Python. For example, the Haskell list comprehension

```
[ i | i <- [1..10], i `mod` 2 == 0]
```

creates the list of all even numbers between 1 and 10 and

```
[ (i, j) | i <- [ e | e <- [1..4], e `mod` 2 == 0],
           j <- [ u | u <- [1..4], u `mod` 2 == 1]]
```

builds the list of all combinations of even and uneven numbers between 1 and 4. The general list comprehensions syntax looks as follows: $[\ h \mid q_1, \ \ldots, q_n\ ]$. The q_i are the qualifiers of the comprehension and each qualifier can either be a *binding* or a *filter*. A binding has the form $b \leftarrow l$ and it introduces the identifier b that is bound to an element of the list l. An identifier that is bound in qualifier q_i is available in all subsequent qualifiers $q_j, j > i$ and also in h, the head of the comprehension. l can reference some previously defined list or be specified as a nested list comprehension. Nested comprehension can reference identifiers of the outer comprehension to create correlated nested comprehensions. A comprehensions can be thought of as a loop nest where each binding creates an additional nesting level and inner loops are in the scope of outer loops.

Final edited form was published in "DBPL 2017: The 16th International Symposium on Database Programming Languages. München 2017", Art. Nr. 7 , ISBN 978-1-4503-5354-0
https://doi.org/10.1145/3122831.3122835


```
-- Haskell
let l = [ (i, j) | i <- [0..10],
                   j <- [i..10],
                   i `mod` j == 0]

// C++
vector<tuple<int, int>> l;
for (int i = 0; i < 10; ++i) {
  for (int j = i; j < 10; ++j) {
    if (i % j != 0) continue;
    l.push_back(make_tuple(i, j));
  }
}
```

Listing 2: Comprehension as a loop nest

Filters are of the form $p(b_x, \ldots, b_z)$. A filter applies a predicate function p to a subset of the available bindings. If the filter evaluates to false the current set of bindings is discarded and the comprehension evaluates the next set of bindings. In terms of a loop nest, a filter is a guard in the outer most level of the nest that contains all bindings of the filter. If the guard evaluates to false it *continues* to the next iteration of *its* loop. The head of a comprehension is an arbitrary function $h(b_x, \ldots, b_z)$ over the comprehension's bindings. In a nutshell, a comprehension evaluates a head function for every set of bindings that passes all filters and inserts the functions's return value into the result of the comprehension. Listing 2 shows a comprehension that pairs all integers between 1 and 10 with their factors. Below the comprehension an analogous C++ code computes the same result using a loop nest.

Wadler [13] discovered that list comprehensions can be generalized to comprehensions over monads with zero. Based on Wadler's definition, we consider a monad with zero to be an abstract data type with four operations that satisfy a small set of algebraic laws. The operations are declared as follows:

- *unit :: a -> Monad a*
- *zero :: Monad a*
- *map :: (a -> b) -> (Monad a -> Monad b)*
- *join :: Monad (Monad a) -> Monad a*

unit inserts a value into a new *Monad a* instance, and *zero* creates an empty *Monad a*, the second-order *map* takes a function from *a* to *b* and turns it into a function from *Monad a* to *Monad b*, and *join* unnests a *Monad (Monad a)* into a *Monad a*. For a discussion of the algebraic monad laws we refer to [13].

Monads provide a generic interface for a variety data structures such as lists, sets, and bags [9]. Wadler introduces four equations that define the semantics of comprehensions in terms of a monad type *t*.

$$[\ h \mid\ ]^t \equiv unit^t\ h, \tag{1}$$

$$[\ h \mid b \leftarrow l^t\ ]^t \equiv map^t(\lambda b.h)\ l, \tag{2}$$

$$[\ h \mid q_1, q_2\ ]^t \equiv join^t([[\ t \mid q_1\ ]^t \mid q_2\ ]^t), \tag{3}$$

$$[\ h \mid b \in \mathbf{Bool}\ ]^t \equiv \text{if b then } [\ h\ ]^t \text{ else } [\ ]^t \tag{4}$$

The equations show that monad comprehensions can operate on many data types that are commonly used in data intensive applications. Monad comprehensions are also naturally composable and nested comprehensions can be correlated to outer comprehensions to express complex data dependencies. We will show in the following sections that monad comprehensions can express the semantics of most of *AL*'s language elements and that they can be automatically translated into efficient execution programs. We are therefore confident that monad comprehensions provide an excellent means to specify the selection, filtering and combination of data for many common query languages.

Mapping al to monad comprehensions. Many elements of *AL*'s query languages can be expressed in terms of monad comprehensions. In this section we are going to introduce semantic translation rules that map *AL* expressions such as relational queries or path patterns onto equivalent monad comprehensions. Details on the implementation of these rules in the *COMP-IR* transformation pass will be discussed in the next section.

We define translation rules as mappings of an abstract translator function **Tr**. Basic relational queries can be translated into a single flat comprehension over bags:

$$\mathbf{Tr}\begin{pmatrix}\text{SELECT}(s_1 \text{ as } n_1, \ldots, s_i \text{ as } n_i)\\ \quad .\text{FROM}(r_1 \text{ as } b_1, \ldots, r_j \text{ as } b_j)\\ \quad .\text{WHERE}(p_1, \ldots, p_k)\end{pmatrix}$$

$$\equiv$$

$$[\ Record(n_1 : \mathbf{Tr}(s_1), \ldots, n_i : \mathbf{Tr}(s_i)) \mid b_1 \leftarrow \mathbf{Tr}(r_1), \ldots, b_n \leftarrow \mathbf{Tr}(r_n), \mathbf{Tr}(p_1), \ldots, \mathbf{Tr}(p_k)\ ]^{Bag}$$

Group by queries are mapped to nested comprehensions:

$$\mathbf{Tr}(\text{SELECT}(s \text{ as } n).\text{FROM}(r \text{ as } b).\text{WHERE}(p).\text{GROUP\_BY}(b.k))$$

$$\equiv$$

$$\big[[Record(n : \mathbf{Tr}(s)) \mid b_2 \leftarrow \mathbf{Tr}(r),\ b_2.k = b_1.k\ ]^{Bag} \mid b_1 \leftarrow \mathbf{Tr}(r)\ \big]^{Bag}$$

Some relational concepts such as aggregations or the *order by* clause can not be easily expressed with monad comprehensions. To support these elements *COMP-IR* provides a set of built-in functions such as *sum*, *avg*, or *sort*. Graph queries are mapped to monad comprehensions using the following rules:

$$\mathbf{Tr}\begin{pmatrix}\text{MATCH}(t_1, \ldots, t_i)\\ \quad .\text{WHERE}(p_1, \ldots, p_j)\\ \quad .\text{RETURN}(s_1 \text{ as } n_1, \ldots, s_k \text{ as } n_k)\end{pmatrix}$$

$$\equiv$$

$$[\ Record(n_1 : \mathbf{Tr}(s_1), \ldots, n_k : \mathbf{Tr}(s_k)) \mid \mathbf{Tr}(t_1), \ldots, \mathbf{Tr}(t_i), \mathbf{Tr}(p_1), \ldots, \mathbf{Tr}(p_j)\ ]^{Bag}$$

$$\mathbf{Tr}(b_1[T_1](p_1) - [l] \rightarrow b_2[T_2](p_2))$$

$$\equiv$$

$$\begin{gathered} b_1 \leftarrow \_vertices,\ b_1.type = T_1,\\ b_2 \leftarrow \_vertices,\ b_2.type = T_2,\\ b_E \leftarrow b_1.edges,\ b_E.label = l,\ b_E.target = b_2 \end{gathered}$$

for path expressions $t \equiv b_1[T_1](p_1) - [l] \rightarrow b_2[T_2](p_2)$. The graph translation rules assume the existence of a single unnamed graph whose physical representation supports vertex centric traversals.

Final edited form was published in "DBPL 2017: The 16th International Symposium on Database Programming Languages. München 2017", Art. Nr. 7 , ISBN 978-1-4503-5354-0
https://doi.org/10.1145/3122831.3122835

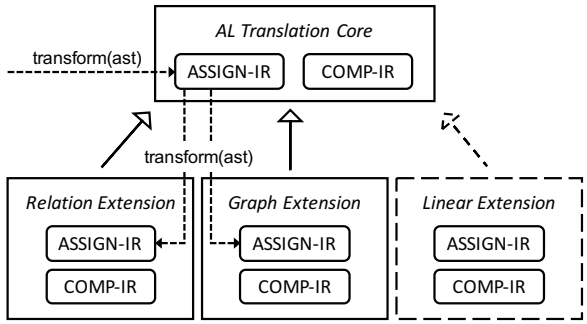

Figure 3: *AL* translator framework

3.2 Translator extensions and Traversals

Extensibility is the guiding design goal for the implementation of the translator framework. *AL* currently supports simple relational and graph pattern matching queries but we plan to extend the language with additional core features and domain expressions in future work. For example, we would like add support for domain specific optimization rules, catamorphism based custom functions [9], or linear algebra expression. Figure 3 shows a conceptual overview of the translator framework. The framework comprises a translation core and several domain specific extensions. Both core and extensions contain transformation components for the two intermediate program representations *ASSIGN-IR* and *COMP-IR*. The extension's components inherit from the core's components so as to enable the composition of transformations using multiple inheritance. The core components transform domain independent language elements such as variable bindings, primitive literals, implicit names, or arithmetic expressions. The translation of domain specific elements is forwarded to available extensions. If all extensions fail to translate an element, compilation is aborted with an error message.

All translator components are implemented as Scala traits[2]. A trait defines a type and can implement methods but it can not define any state. Due to the lack of state, traits can be involved in multiple inheritance without causing anomalies when the inheritance hierarchy contains multiple copies of the same type. Listing 3 shows excerpts of the traits that define the core *ASSIGN-IR* transformation. The listing contains the three traits *AssignBase*, *LiteralsToAssign*, and *BoolExpToAssign*, and the singleton object *AstToAssign*. Both *LiteralsToAssign* and *BoolExpToAssign* inherit from *AssignBase* and *AstToAssign* inherits from all three traits. At the end of the listing, object *AstToAssign* is used to transform an AST object into *ASSIGN-IR* form.

Trait *AssignBase* defines the method *toAssign* which translates a Scala AST object into *ASSIGN-IR* form. In *AssignBase*, *toAssign* simply throws an exception to abort the translation. This represents the base case where the AST did not match any translation rules. Traits *LiteralsToAssign* and *BoolExpToAssign* override *toAssign* and use pattern matching to find semantic elements that can be mapped to *ASSIGN-IR* nodes. If a rule matches successfully, it returns an

[2]http://docs.scala-lang.org/tutorials/tour/traits.html

```
trait AssignBase extends Transformation {
  var bindings = Map.empty[Tree, Binding]

  override def transform(t: Tree): Binding =
    findBinding(t) match {
      case Some(b) => b
      case None => createBinding(t, toAssign(t))
    }

  def createBinding(t: Tree, n: AssignNode) = {
        bindings += t -> Binding.fresh(n)
        bindings(t)
  }

  def toAssign(t: Tree): AssignNode
        throw Exception("Could not transform: " + t)
}

trait LiteralsToAssign extends AssignBase {
  override def toAssign(t: Tree): AssignNode =
    t match {
      case Literal(Constant(i: Int)) =>
        IntLiteral(i)
      // ...
      case _ =>
        super.transform(t) // delegate to super
    }
}

trait BoolExpToAssign extends AssignBase {
  override def toAssign(t: Tree): AssignNode =
    t match {
      case q"$lhs && $rhs" =>
        And(transform(lhs), transform(rhs))
      // ...
      case _ =>
        super.transform(t) // delegate to super
}

// create an ast to ASSIGN-IR transformer
object AstToAssign extends AssignBase
                   with LiteralsToAssign
                   with BoolToAssign

// transform some t: Tree into ASSIGN-IR
val assignForm = AstToAssign.transform(t)
```

Listing 3: *ASSIGN-IR* transformation

object that represents one of the *ASSIGN-IR* functions such as *And* or *IntLiteral*. *AssignBase* further defines the method *transform* which essentially calls *toAssign* and stores the result in a new variable binding. Translation rules, such as the one for boolean conjunction, recursively call *transform* on AST subtrees and thereby transform the whole AST.

Both *LiteralsToAssign* and *BoolExpToAssign* define a default rule that delegates translation to *super*. This is the key technique that enables composition in the *AL* translator. The technique relies on Scala's *type linearization* rules which define how delegation to *super* is handled with regard to multiple inheritance. Consider a type *D* that extends the types *A*, *B*, and *C*, in that order. If a method in *D* delegates to *super*, method resolution first searches type *C*, than *B*, and finally *A* for a definition of the function. That is, method resolution traverses the list of extended types from right to left

Final edited form was published in "DBPL 2017: The 16th International Symposium on Database Programming Languages. München 2017", Art. Nr. 7 , ISBN 978-1-4503-5354-0
https://doi.org/10.1145/3122831.3122835

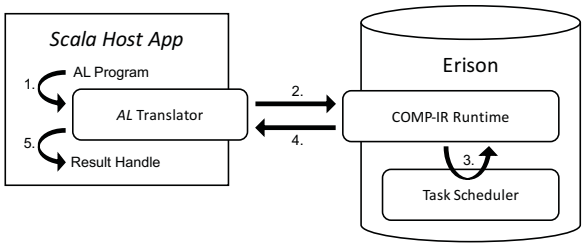

Figure 4: Erison *COMP-IR* architecture

to find a definition[3]. With regard to *AstToAssign*, this means that the definitions of *toAssign* of all three traits are composed into a sequence of pattern matching rules that is sequentially evaluated until a match is found. The sequence starts with the rules of *BoolToAssign*, continues with the rules of *LiteralsToAssign*, and ends with the exception throwing implementation of *AssignBase*. Extending *AL* is therefore simply a matter of defining additional traits and adding them to the inheritance chains of the *ASSIGN-IR* and *COMP-IR* transformations.

The *COMP-IR* transformation component also uses multi trait inheritance as a means for extensibility and pattern matching to identify program elements. However, the transformation algorithm itself is quite different compared to the *ASSIGN-IR* approach. For a start, the entrance point of the transformation is the last variable binding that has been generated by the *ASSIGN-IR* transformation, which is the return value of the *AL* program. In other words, the *COMP-IR* transformation starts at the final statement of the *ASSIGN-IR* program and recursively works its way backwards. It immediately follows that the transformation will only consider statements that are reachable from the return value of the program.

As we have discussed in Section 2, *AL* programs have to return values of a certain set of types. The *COMP-IR* transformation asserts this property by matching the programs return value against a set of so called *trigger combinators*. This includes combinators such as *SqlWhere* or *GraphReturn*. If the transformation finds a trigger, it saves it in an internal buffer, and switches into a domain specific mode. If, for example, the trigger is a *SqlWhere*, the transformation switches into *SQL* mode, and if it is a *GraphReturn*, it switches into *GraphMode*. The combination of a combinator buffer and a transformation mode is called a translation context. Once the transformation has switched into domain mode, it recurses over the arguments of the buffered trigger combinator and uses mode specific matching rules to identify further *ASSIGN-IR* combinators that belong to the same query. Every combinator that is found in this way is also inserted into the buffer. Eventually, a complete query has been matched and at that point the transformation simply uses a translation rule similar to the ones we have discussed in Section 3.1 to translate the buffered combinators into a comprehension. Ultimately, that comprehension is returned as the result of the transformation. If at any point the transformation encounters an *ASSIGN-IR* combinator that can not be matched in the current mode, the transformation pushes a nested translation context that starts in trigger detection mode and it tries to build a nested comprehension. Not surprisingly, additional triggers and modes can be introduced by inheriting from additional traits.

[3]Scala's actual linearization rules are somewhat more involved as they also have to consider super types in the extension list. However, four our purposes the simplified rules are sufficient.

4 *COMP-IR* RUNTIMES

To be of any practical use, *COMP-IR* programs have to be translated into executable programs at some point. *COMP-IR* is an abstract representation that supports different approaches to accomplish that translation. In [5], Alexandrov et al. demonstrate how comprehensions can be mapped onto the processing operators of the big data processing framework Apache Flink[4]. Similarly, we were able to create an early prototype of a *COMP-IR* Flink runtime that can execute *COMP-IR* programs. Essentially, a *COMP-IR* Flink runtime consists of a catalog that maps logical names to physical data objects and an algorithm that translates comprehensions onto Flink operators.

In this paper however, we want to explore the translation of *COMP-IR* into efficient programs for shared memory multi processor systems. For this purpose, we created the *Erison* data processing engine and a corresponding *COMP-IR* runtime. *Erison* is written entirely in C++ and uses standard data structures for data storage. Data is stored in tables in a column oriented format. The engine uses the Intel© TBB task scheduler [3] for parallelized processing. The scheduler assigns user provided processing tasks to a set of worker threads and relies on a work stealing mechanism for load balancing.

Figure 4 shows the system architecture of the Scala host application and the *Erison* execution engine. A complete roundtrip through the system involves five steps. (1) the Scala host application submits an *AL* string to one of the translator's compilation methods, (2) the translator transforms the program into *COMP-IR*

[4]https://flink.apache.org/

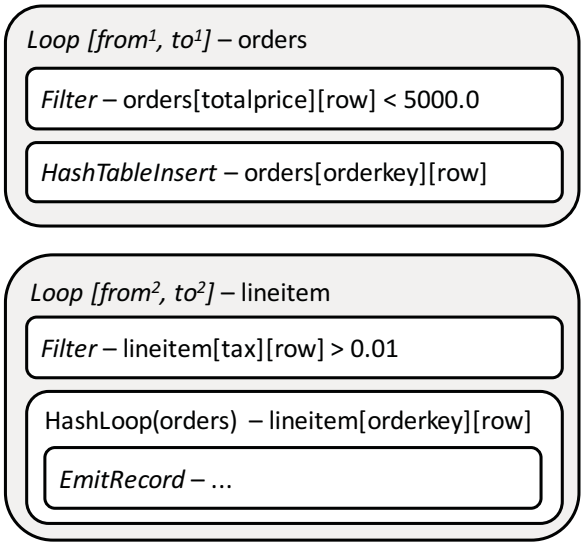

Figure 5: Hash join program

Final edited form was published in "DBPL 2017: The 16th International Symposium on Database Programming Languages. München 2017", Art. Nr. 7 , ISBN 978-1-4503-5354-0
https://doi.org/10.1145/3122831.3122835

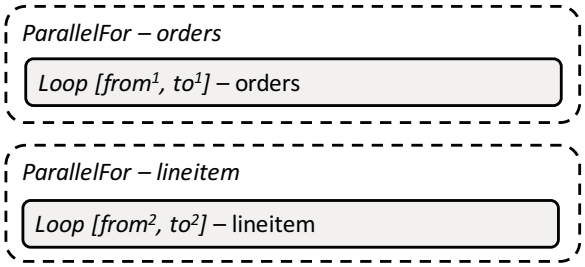

Figure 6: Parallelized program

representation and sends the resulting JSON object to the *Erison COMP-IR* runtime, (3) the runtime uses program interpretation and runtime code generation to create tasks for the scheduler, (4) the runtime stores the result of the *COMP-IR* program in a permanent data object and sends an object handle to the *AL* translator, (5) the translator returns the handle to the host application which can use it to retrieve program results.

The *Erison COMP-IR* runtime translates a *COMP-IR* program into a sequence of possibly nested loops. Each loop traverses over a range of a table, applies filters, and eventually executes a core statement which can be either another (nested) loop or a materialization operation. Figure 5 shows the loop sequence for a hash join on *orders* and *lineitem*. The first loop traverses the *orders* table and filters all rows whose *totalprice* column has a value below 5000. Its core is a *HashTableInsert* statement which materializes the *orderkey* column in a hashtable. The second loop traverses the *lineitem* table and filters rows whose *tax* value is greater than 0.01. Its core statement is a nested *HashLoop* that queries the *orders* hash table with key *lineitem[orderkey][row]*. The *HashLoop* does not apply filters and its core is an *EmitRecord* statement which materializes join results. The fixed loop structure with a single materialization operation guarantees a minimal number of materialized intermediate results.

The individual loop nests are dynamically generated and compiled using LLVM. This ensures maximum performance in the critical sections of the program. Each loop nest is further embedded in a *ParallelFor* structure as depicted in figure 6. *ParallelFor* is a task scheduler function that partitions a numeric range and calls a user defined function on each range partition in parallel. The *ParallelFor* structure builds a nest that mirrors the nesting of the generated loop nest. That is, a generated loop nest over tables *A* and *B* is embedded in a *ParallelFor* nest over *A* and *B*.

COMP-IR programs are translated into loop sequences using a simple algorithm. First, a comprehension creates a single nested loop with one nesting level for each binding of the comprehension. Second, the comprehension's filter expressions are added to the loop nest and depending on the type of the filter, the loop nest is split into two consequent nests that are executed sequentially. This is true, for example, for an equality filter on integer fields of two bindings which can be implemented as hash join. Finally, the the comprehensions head expressions is inserted into the last nest of the sequence.

5 RELATED WORK

The primary goal of *AL* is to provide a programming language that integrates several domain specific query- and data processing concepts into a single coherent programming model. The primary goal of *COMP-IR* is to define a domain independent program intermediate representation that can be translated into efficient programs for modern hardware and system architectures. In combination, *AL* and *COMP-IR* create a programing environment that is both flexible in the applications that it supports and in the execution environments that it can target.

Others have recognized the need for such a system as well. To that end, Duggan et al. propose the *BigDAWG* polystore system [6]. BigDAWG is a multi-domain data processing system that delegates workloads to a set of specialized database engines. For example, it can connect to multiple RDMS that each offer different non functional properties and delegate each query to the engine that is best equiped to process it. BigDAWG's programming language allows the composition of SQL and array database queries in a single program. During execution the system splits up the program and forwards the individual queries to an engine that can handle them. This approach eliminates the need for a dedicated query compilation and allows to make the best use of the connected engines. Unfortunately, the limited abstraction from specific query languages also voids any approach to extend BigDAWG's language with additional features or domain concepts. In addition, the lack of a domain independent intermediate representation makes it hard to add support for general purpose processing engines, such as Appache Spark [2], without having to implement a complete set of query compilers.

Musketeer [7] is a system that can translate different existing query languages into workflows for a set of dataflow engines. The compilation relies on a graph based, turing complete intermediate representation which provides a common ground for query languages and execution engines. Musketeer supports input languages such as *Hive*, *Giraph*, or *SparkSQL* and can generate programs for Spark, Hadoop, and similar engines. However, Musketeer does not offer a language that integrates multiple application domains, such as *AL*.

Weld [11] uses a flexible program intermediate representation and a central data processing engine to avoid costly copying of datasets between various libraries. The program intermediate representation is based on nested loops and closely resembles monad comprehensions. The authors argue that a loop oriented representation is superior to combinator based representations because it facilitates important optimization such as loop fusion and vectorization. Weld underlines the performance penalty that is implied by using multiple data processing systems as compared to using a single one. In addition, Weld provides another example for a data oriented program intermediate representation that can capture a large variety of applications.

6 CONCLUSIONS AND FUTURE WORK

In this article we present our approach to a data oriented programming environment for multiple application domains. Our programming language *AL* supports relational queries, graph pattern matching queries, and also the composition of these queries. We have

Final edited form was published in "DBPL 2017: The 16th International Symposium on Database Programming Languages. München 2017",
Art. Nr. 7 , ISBN 978-1-4503-5354-0
https://doi.org/10.1145/3122831.3122835

presented an extensible translation framework that can transform *AL* programs into the domain and platform independent *COMP-IR* program intermediate representation and we have shown how *COMP-IR* programs can be compiled into workloads for our experimental data processing engine *Erison*.

AL and *COMP-IR* are still in early development and a lot of interesting questions remain. An important open area are program optimizations and especially optimizations that cross the boundaries between different query models. Related to that, we plan to explore which optimizations can be done during the transformation into *COMP-IR* form and which ones have to be delegated to the *COMP-IR* runtime. Besides these issues, we are also interested in extending *AL* itself. On the one hand, we want to add a linear algebra sublanguage. This will allow us to explore the interactions between different data types, namely bags of records and fixed sized matrices. On the other hand, we are interested in adding support for the definition of custom functions in *AL*.

REFERENCES

[1] [n. d.]. http://neo4j.com/docs/developer-manual/current/cypher/. ([n. d.]).
[2] [n. d.]. http://spark.apache.org/. ([n. d.]).
[3] [n. d.]. https://www.threadingbuildingblocks.org/. ([n. d.]).
[4] Divy Agrawal, Sanjay Chawla, Ahmed K Elmagarmid, Zoi Kaoudi, Mourad Ouzzani, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Nan Tang, and Mohammed J Zaki. 2016. Road to Freedom in Big Data Analytics.. In *EDBT*. 479–484.
[5] Alexander Alexandrov, Andreas Kunft, Asterios Katsifodimos, Felix Schüler, Lauritz Thamsen, Odej Kao, Tobias Herb, and Volker Markl. 2015. Implicit parallelism through deep language embedding. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 47–61.
[6] Jennie Duggan, Aaron J Elmore, Michael Stonebraker, Magda Balazinska, Bill Howe, Jeremy Kepner, Sam Madden, David Maier, Tim Mattson, and Stan Zdonik. 2015. The bigdawg polystore system. *ACM Sigmod Record* 44, 2 (2015), 11–16.
[7] Ionel Gog, Malte Schwarzkopf, Natacha Crooks, Matthew P Grosvenor, Allen Clement, and Steven Hand. 2015. Musketeer: all for one, one for all in data processing systems. In *Proceedings of the Tenth European Conference on Computer Systems*. ACM, 2.
[8] Torsten Grust. 2000. Comprehending queries. In *Ausgezeichnete Informatikdissertationen 1999*. Springer, 74–83.
[9] Torsten Grust. 2004. Monad comprehensions: a versatile representation for queries. In *The Functional Approach to Data Management*. Springer, 288–311.
[10] Torsten Grust and Marc H Scholl. 1999. How to comprehend queries functionally. *Journal of Intelligent Information Systems* 12, 2-3 (1999), 191–218.
[11] Shoumik Palkar, James J Thomas, Anil Shanbhag, Deepak Narayanan, Holger Pirk, Malte Schwarzkopf, Saman Amarasinghe, Matei Zaharia, and Stanford InfoLab. 2017. Weld: A common runtime for high performance data analytics. In *Conference on Innovative Data Systems Research (CIDR)*.
[12] Michael Stonebraker and Ugur Cetintemel. 2005. "One size fits all": an idea whose time has come and gone. In *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*. IEEE, 2–11.
[13] Philip Wadler. 1990. Comprehending monads. In *Proceedings of the 1990 ACM conference on LISP and functional programming*. ACM, 61–78.