A Multi-target, Multi-paradigm DSL Compiler for Algorithmic Graph Processing

Houda Boukham
Ecole Mohammadia d'Ingénieurs
Rabat, Morocco
Oracle Labs
Casablanca, Morocco

Guido Wachsmuth
Oracle Labs
Zurich, Switzerland

Martijn Dwars
Oracle Labs
Zurich, Switzerland

Dalila Chiadmi
Ecole Mohammadia d'Ingénieurs
Rabat, Morocco

Abstract

Domain-specific language compilers need to close the gap between the domain abstractions of the language and the low-level concepts of the target platform. This can be challenging to achieve for compilers targeting multiple platforms with potentially very different computing paradigms. In this paper, we present a multi-target, multi-paradigm DSL compiler for algorithmic graph processing. Our approach centers around an intermediate representation and reusable, composable transformations to be shared between the different compiler targets. These transformations embrace abstractions that align closely with the concepts of a particular target platform, and disallow abstractions that are semantically more distant. We report on our experience implementing the compiler and highlight some of the challenges and requirements for applying language workbenches in industrial use cases.

CCS Concepts: • **Software and its engineering → Domain specific languages**.

Keywords: domain-specific languages, multi-paradigm compilers, intermediate representation

ACM Reference Format:
Houda Boukham, Guido Wachsmuth, Martijn Dwars, and Dalila Chiadmi. 2022. A Multi-target, Multi-paradigm DSL Compiler for Algorithmic Graph Processing. In *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering (SLE '22), December 06–07, 2022, Auckland, New Zealand*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3567512.3567513

SLE '22, Dec 05–10, 2022, Auckland, New Zealand

ACM ISBN 978-1-4503-9919-7/22/12…$15.00
https://doi.org/10.1145/3567512.3567513

1 Introduction

Domain-specific languages (DSLs) provide high-level abstractions that are closely aligned with the domain for which the language is built [40]. DSL compilers bridge the gap between these abstractions and the low-level concepts of the target platform. This becomes more challenging for compilers targeting multiple platforms with significantly different characteristics. We face this challenge in our work on a compiler for *PGX Algorithm*, a DSL for algorithmic graph processing [25]. At its core, *PGX Algorithm* provides several abstractions for concurrent iterations over vertices and edges of a graph. The compiler is available as an option in several industrial products and an active research project. It currently targets four different platforms:

- a Java-based graph-processing runtime, where the graph is loaded into the memory of a single machine,
- a distributed graph-processing runtime implemented in C++, where the graph is distributed over multiple machines,
- a relational database, where the graph is stored in database tables and can be processed in PL/SQL, and
- an experimental graph algorithm engine targeted at in-memory columnar databases with transactional semantics.

These target platforms do not only differ in their programming languages, but also in their underlying paradigms for supporting concurrent iteration. The single-machine runtime relies on a parallel runtime system for multi-socket single-machines [10]. The distributed runtime relies on a distributed iteration abstraction that offers message sending and receiving capabilities [12]. In the relational database, SQL queries iterate concurrently over table rows [34], while the in-memory engine can make use of low-level multi-threading mechanisms of the database.

Despite these differences, we see common patterns in the compiler backends and in the generated code. For example, the compiler generates top-level structures for each concurrent iteration occurring in a graph algorithm. While the

details of these structures depend heavily on the target platform, they share a common skeleton. The compiler backends use similar transformations to collect concurrent iterations from the graph algorithm and to create the corresponding top-level structures. Furthermore, the compiler uses common transformations such as procedure extraction and subexpression extraction for all target platforms. Common optimizations such as dead code elimination can generally be applied in an early compilation stage, regardless of the backend in question. Other optimizations such as loop fusion can benefit some, but not all, target platforms.

In this paper, we present a multi-target, multi-paradigm architecture for our compiler. This architecture embraces similarities in the transformations and the optimizations required for different target platforms. It supports a modular compiler implementation, allowing us to reuse transformations and optimizations, and to compose them as needed. This modularity also makes it possible to add support for additional target platforms without breaking or changing the implementation for already supported target platforms.

Our approach centers around an intermediate representation (IR) at the level of which we implement reusable, composable transformations to be shared between the different compiler targets. These transformations ensure certain forms of code designed to embrace abstractions that align with the concepts of a particular target platform, and disallow abstractions that are semantically distant. They are then composed into a multi-stage transformation, breaking down the semantic gap towards the target platforms into several smaller gaps bridged by these transformation [40].

Our contributions for this paper are:

- an IR for algorithmic graph processing (Section 2),
- a set of reusable, composable transformations over this IR (Section 3),
- an experience report on implementing the *PGX Algorithm* compiler with Spoofax [15], where we highlight some of the the challenges and requirements that language workbenches need to meet in order to be suitable for industrial use (Section 4).

We conclude the paper with a discussion of related and future work in Sections 5 and 6.

2 An Intermediate Representation for Algorithmic Graph Processing

At the core of our compiler, we introduce an IR for algorithmic graph processing.

The design of this IR was informed by our previous work on a compiler for *Green-Marl*, a DSL for algorithmic graph processing [11]. This compiler initially targeted only the Java-based graph processing runtime. We implemented the compiler with multi-stage transformations to bridge the gap between the DSL and this target platform. At several stages, the compiler enhanced the representation of an algorithm with additional information, which could then be used in subsequent transformations. This fixed the order of transformations to some degree, since each transformation worked only on a particular representation. While we were able to re-order transformations that were applied on the same representation, it was harder to re-order transformations that were applied on different representations. This turned out to be problematic when we extended the compiler to target the distributed graph processing runtime in C++, which presents a different computing paradigm for concurrent iteration. We could not simply reuse existing transformations, since we needed to apply them in a different order, requiring a different representation than the one handled by the transformation. We also needed new representations that would explicitly capture details of the distributed target platform. We ended up with a non-modular implementation, where similar transformations were implemented multiple times over different representations and where changes in transformations tend to influence the compilation for both target platforms.

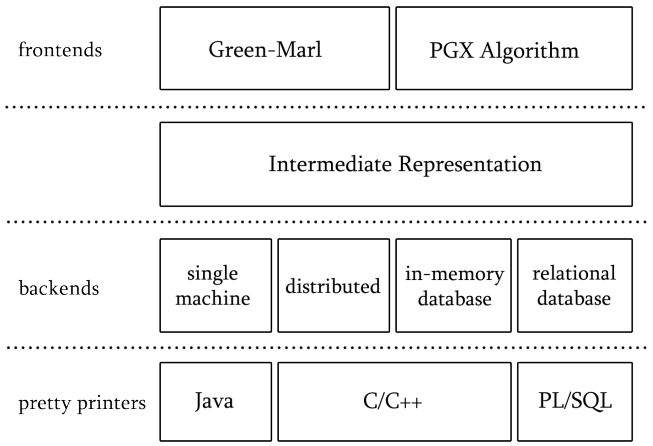

Figure 1. Components of the *PGX Algorithm* Compiler.

Based on this experience, we now support a modular compiler architecture, which separates frontend components, intermediate representation, and backend components for various target platforms, as depicted in Figure 1. The IR presented in this section is at the center of this architecture. Transformations over this IR are shared across different backend components without breaking existing implementations. As a consequence, backend components are mutually independent and changes in one backend do not affect other components.

We introduce the IR with a small example algorithm, before we introduce its syntax in detail in the remainder of the section. We mainly care about the abstract syntax of the IR, but provide example programs in concrete syntax for readability.

Figure 2 shows a simplified version of the infamous PageRank algorithm [26]. The algorithm takes a graph as an input

```
1 procedure void pagerank(in graph G,
2   out property<vertex(G), double> rank
3 ) {
4   long N;
5   double diff;
6   property<vertex(G), double> newRank;
7   double x;
8   N = numVertices(G);
9   setProperty(G, rank, 1d/(double) N);
10  x = 0.15d/(double) N;
11  do {
12    diff = 0.0d;
13    foreach(v: V(G)) (true) {
14      double inSum =
15        sum(w -[e]-> _: E(G, v)) (true) {
16          let
17            double rankVal = w.rank;
18            long degree = outDegree(G, w);
19          in
20            rank/(double) degree
21        };
22      double newRankVal = x+0.85d*inSum;
23      double rankVal = v.rank;
24      diff += |newRankVal-rankVal|;
25      v.newRank = newRankVal;
26    }
27    copyProperty(G, rank, newRank);
28  } while (diff>0.001d);
29 }
```

Figure 2. PageRank algorithm expressed in IR.

(1)	p	=	$td\ td^*$
(2)	td	=	**procedure** $rt\ id$ **(** pd^* **)** b
(3)	pd	=	$pt\ id$
(4)	pt	=	$a\ t$
(5)	a	=	**in** | **out** | **in** − **out**
(6)	t	=	**int** | **long** | **float** | **double** | **number**
		|	**bool** | **string**
(7)	rt	=	**void** | t
(8)	ft	=	$pt^* \rightarrow rt$
(9)	b	=	**{** $vd^*\ s^*\ r$ **}**
	vd	=	$t\ id$;
(10)	s	=	**if (** e **)** b **else** b
		|	**while (** e **)** b | **do** b **while (** e **)** ;
		|	lr **=** e ;
(11)		|	lr **=** tr **(** e^* **)** ; | tr **(** e^* **)** ;
	r	=	**return** e ; | **return**; | ε
(12)	e	=	n | s
(13)		|	e **+** e | e **−** e | e **∗** e | e **/** e | e **%** e | **−** e | **|** e **|**
(14)		|	**(** t **)** e
(15)		|	e **<** e | e **≤** e | e **>** e | e **≥** e | e **=** e | e **≠** e
(16)		|	**true** | **false** | e **∧** e | e **∨** e | **¬** e
(17)		|	lr
(18)	lr	=	**[** a **param** t **]** id | **[local** t **]** id | id
(19)	tr	=	**[procedure** ft **]** id | **[builtin** ft **]** id | id
	id		names
	n		numeric literals
	s		string literals

Figure 3. Syntax of imperative core constructs of the IR.

parameter and calculates the rank of each vertex in an output parameter (line 2). Similar to *Green-Marl*, the IR adopts the *property graph model*, in which vertices and edges can be associated with arbitrary *properties* as key-value pairs. The algorithm relies on a vertex property to store the rank of each vertex. After initializing the rank property to the inverse of the number of vertices in the graph (lines 8, 9), the algorithm performs a fixpoint iteration until the aggregated error between new and old rank values hits a threshold (lines 11-28). In each iteration, the algorithm iterates over the vertices of the graph (lines 13-26), aggregates a sum over the incoming neighbors of each vertex (lines 14-21), calculates the new rank value for each vertex (line 22), and aggregates the difference between old and new rank values (line 24). Finally, new values are copied into the rank property (line 27).

Imperative Core Constructs. At its core, the IR resembles procedural, imperative, general-purpose programming languages. Figure 3 provides a syntax definition for the imperative core of the IR. In our metasyntax notation, we print *sorts* in italics and *reserved words* in bold. Sorts are defined with the symbol =. Alternative definitions are separated by the choice operator |. The $*$ operator indicates zero or more repetitions.

A program p consists of a main procedure and additional local procedures, which are declared at the top-level (1). There is no syntactic distinction between main and local procedures: the first top-level declaration is considered to be the main procedure and any following top-level declarations are considered local procedures. A top-level declaration td specifies the return type, the name, the parameters, and a block of statements of a procedure (2). Each parameter declaration pd includes explicit access information, the type of the parameter, and its name (3, 4). We distinguish **in** parameters (read-only), **out** parameters (write-only), and **in** − **out** parameters (5).

The IR provides primitive types for numbers (6), booleans, and strings. For procedures without a return value, **void** can be used as a return type (7). We rely on function types (8) to represent the types of top-level declarations in semantic analysis and in annotated references to top-level declarations (19).

The syntax for statements aims to minimize syntactic variants, reducing the number of cases transformations have to cover. Variables have to be declared at the start of a block, while return statements can only occur at the end of a block (9). Blocks are required as child nodes in control-flow statements (10).

Expressions include numeric and string literals (12), numeric expressions (13), explicit casts (14), comparisons (15), boolean expressions (16), and references to local parameters and variables (17). We disallow procedure calls in expressions, as they might have output parameters, which complicates compilation. Instead, we provide dedicated statements for procedure calls (11).

References to parameters, local variables, and top-level declarations can be optionally annotated with static semantics information. These annotations can be used to make semantic analysis results explicit, such that they can be employed and maintained in transformations without the need for semantic re-analysis. Annotations for local references capture whether a variable refers to a parameter or a local variable and indicate its type information (18). In the case of a parameter reference, the annotation contains additional information indicating the parameter's access information. Annotations for calls to top-level declarations capture the kind of top-level declaration and type information (19).

Domain-Specific Constructs. We extend the imperative core constructs of our IR with domain-specific constructs for graph processing, namely graph-related types, statements for iterations over graph elements, and expressions for property access and aggregations over graph elements.

The clear separation between core and graph constructs simplifies compilation, since core constructs can directly be mapped to their semantic equivalents, while graph constructs often require lowerings before being translated to the target languages. Figure 4 shows the syntax definition for the IR graph constructs. It uses the same notation as Figure 3. For sorts already defined in the core grammar, we provide additional alternative definitions (. . .). These extensions do not create ambiguities in concrete syntax, with the exception of the block sort, which is easy to disambiguate in favor of the new construct.

Our IR supports four kinds of types related to property graphs: graph types, graph element types for vertices and edges, property types, and collection types. Graph types (1) can only be used for input parameters in top-level declarations. Graph element types (2, 5) explicitly include the graph they belong to. Property types (3) include two type parameters: the first parameter specifies the graph element type the property is associated with (that is, whether it is a vertex property or an edge property), and the second parameter specifies the type of the data stored in the property. Similarly, graph collection types (4) include a single type parameter, specifying the type of elements in the collection. Collections can only contain vertices or edges of a graph.

We extend the block construct with instructions for memory allocation (7) and de-allocation (8) of graph properties and collections (6). These instructions surround the block's list of statements.

```
        t    =  ...
(1)          |  graph | directed graph | undirected graph
(2)          |  et
(3)          |  property < et , t >
(4)          |  set < et > | sequence < et >
             |  order < et > | stack < et >
             |  queue < et >
(5)     et   =  vertex ( id ) | edge ( id )
(6)     b    =  ... | { vd* am* s* fm* r }
(7)     am   =  allocate lr;
(8)     fm   =  free lr;
(9)     s    =  ... | pr = e ; | lr = ge ; | gi
(10)    pr   =  [ a param t ] id.id | [ local t ] id.id | id.id
(11)    ge   =  pr
(12)         |  [ t ] op ce { le } | op ce { le }
        op   =  sum | product | max | min | average | any | all
(13)    ce   =  [ te ] ( ee ) ( le ) | ( ee ) ( le )
(14)    te   =  id : VT ( id ) | id --id--> id : ET ( id )
             |  _ --id--> id : ET ( id , id )
             |  id --id--> _ : ET ( id , id )
(15)    ee   =  id : V ( id ) ] | id --id--> id : E ( id )
             |  _ --id--> id : E ( id , id ) | id --id--> _ : E ( id , id )
(16)    le   =  let ld* in e
(17)    ld   =  t id = e ; | t id = tr ( e* ); | t id = ge ;
(18)    gi   =  foreach ce { ld* is* }
             |  inBFS ce { ld* is* } inReverse ( le ) { ld* is* }
(19)    is   =  pr = e ; | pr aop e ; | lr aop e ; | gi
             |  if ( le ) { ld* is* } else { ld* is* }
        aop  =  + = | * = | min = | max = | ∧ = | ∨ =
(20)    td   =  ...
             |  iteration id ( pd* ) { gi }
             |  aggregation rt id ( pd* ) { return ge;}
(21)    tr   =  ... | [ iteration ft ] id | [ aggregation ft ] id
```

Figure 4. Syntax of the graph constructs of the IR.

In addition to core expressions, our IR offers two kinds of expressions (11) to access properties (10) and to aggregate values from a graph (12). We intentionally separate graph-related expressions from core expressions. For that, we provide dedicated statements for assigning the value of a graph expression to a variable (17). We also use **let** expressions (16) to separate core and graph expressions in filter expressions and output expressions (*e.g.* inside an aggregation). Local declarations in a **let** expression assign graph expressions (aggregations and property references) to local variables, which can then be used in the body of the **let** expression, thus ensuring the body is a core expression. Local variable declarations can also assign core expressions or calls to procedures of non-void return types.

let expressions can refer to all variables declared in their syntactic parents such as surrounding aggregations or surrounding block statements. The visibility of variable declarations in **let** expressions is the same as in so-called **let***

expressions in Lisp dialects. A variable is visible in subsequent declarations of the **let** expression and in the body of the expression. Thus, a variable can be used in the right-hand sides of subsequent declarations and in the body of the **let** expression. Variables must not shadow any other variables.

The IR supports parallel iterations over graphs through **foreach** loops and **inBFS** traversals (18). These iterations can either be inlined within the procedure as statements, or extracted outside of the procedure's scope as top-level declarations as we will elaborate on later.

In addition to element iterators (15), which iterate over a set of vertices, edges, or neighbors of a vertex, the IR syntax allows for optional table iterators (14) for iterating over multiple vertex and edge tables. This is to match the tabular representation of graphs in the shared-memory and database runtimes. When present, the table iterator *te* follows the syntax of the element iterator *ee*. In a parallel iteration, variables bound in iterators (13) are visible throughout the whole loop and must not shadow any other variables.

The filter expression is mandatory. Complex filter expressions typically need to be expressed as a **let** expression.

A loop block (block of a **foreach** loop or **inBFS** traversal) consists of a list of local variable declarations (17), followed by a list of iterator statements (ref. (19). As with **let** expressions, local variable declarations assign graph expressions (aggregations and property references) to local variables, such that only core expressions appear in the right-hand side of iterator statements, namely property updates and reduction assignments.

Loop blocks can refer to all variables declared in their syntactic parents such as surrounding iterator statements or surrounding block statements. A local variable declared in a loop block is visible in subsequent declarations of the loop block and in its iterator statements. Thus, a variable can be used in the right-hand sides of subsequent declarations. Variables must not shadow any other variables. Variable declarations in loop blocks are final. The variables declared in a loop block cannot be re-assigned in its iterator statements. Iterator statements are property updates, reduction assignments, and nested **foreach** loops.

Our compiler backends often need to extract parallel iterations outside the scope of the procedure the iterator is used in before translating it to the corresponding language construct. To support this common feature, the IR supports top-level declaration of iterations and aggregations (20). These declarations can then be invoked through procedure calls at their corresponding positions in the algorithm. Annotations for calls to iterations and aggregations help to distinguish them from ordinary procedure calls (21).

3 Transformations over the IR

Backends require certain transformations and optimizations to be applied to the program before transforming it to a lower intermediate representation, or generating target code. Many of these transformations can be shared between two or more backends, but the order in which they are called may differ. We want to implement these transformations such that they can be composed, and reused efficiently between backends. For that, we rely on several *normal forms*.

Table 1. Normal forms required by each backend.

	Single Machine	Distributed	Relational Database	In-Memory Database
Single-issue iterations			•	
Top-level iterations	•	•	•	•
Explicit semantic information	•	•	•	•
Explicit table iterators	•		•	
Explicit memory management	•		•	•
Privatization	•	•		•

We consider a normal form a subset of a language with certain syntactic restrictions. For every program, there exists at least one semantically equivalent program in normal form. A program may conform to more than one normal form simultaneously. Normal forms allow us to embrace abstractions that align closely with the concepts of a particular target platform, and disallow abstractions that are semantically more distant.

Table 1 shows the normal forms used in our compiler and indicates which backends require them. We now discuss each of these normal forms in detail.

Single-Issue Iterations. In this normal form, iterations are limited to either update a single property or to compute an aggregation. Iterations can only combine property updates and aggregations, if the aggregation is nested within the property update, and its result is used to update the property. This normal form is mainly required by the PL/SQL backend, whose target platform can only perform one property update per parallel iteration.

Figure 5 shows the same PageRank algorithm from Figure 2, but with the main **foreach** loop split into an iteration updating the new rank property (lines 13- 23), and an aggregation computing the difference between the old and new rank values (lines 24- 30), in accordance with the single-update normal form.

The transformation to establish this normal form is a top-down traversal which replaces iterations that contain multiple statements with multiple iterations that contain a single statement each.

Top-level Iterations. In this normal form, iterations can only occur as top-level declarations, and cannot occur inside

```
1 procedure void pagerank(in graph G,
2   out property<vertex(G), double> rank
3 ) {
4   long N;
5   double diff;
6   property<vertex(G), double> newRank;
7   double x;
8   N = numVertices(G);
9   x = 0.15d/(double) N;
10  setProperty(G, rank, 1d/(double) N);
11  do {
12    diff = 0.0d;
13    foreach(v: V(G)) (true) {
14      double inSum =
15        sum(w -[e]-> _: E(G, v)) (true) {
16          let
17            double rankVal = w.rank;
18            long degree = outDegree(G, w);
19          in
20            rankVal/(double)degree
21        };
22      v.newRank = x + 0.85d*inSum;
23    }
24    diff = sum [V_t: VT(G)] (v: V(V_t)) (true) {
25      let
26        double newRankVal = v.newRank;
27        double rankVal = v.rank;
28      in
29        |newRankVal-rankVal|
30    };
31    copyProperty(G, rank, newRank);
32  } while (diff > 0.001d);
33 }
```

Figure 5. PageRank algorithm with single-issue iterations.

procedure bodies. Instead, top-level iterations and can be called at the corresponding positions in the procedure. This normal form aims to decrease overhead and simplify backend implementations, as most backends require iterations to be extracted in order to generate corresponding top-level structures. The single machine backend, for example, generates a Java class for each iteration, whereas the in-memory backend generates a number of C structures and callbacks.

The PageRank variant in Figure 6 adheres to this normal form by moving the main **foreach** loop outside of the main procedure's scope and wrapping it in a top-level iterator declaration (lines 18, 38), and then calling this iteration at its previous position in the algorithm (line 13).

The transformation to establish this normal form traverses the program in a top-down traversal, collects parallel iterations, creates top-level declarations for each one, and replaces the statement with a call to the corresponding declaration.

Explicit Semantic Information. In this normal form, all variable references, calls, and aggregations need to provide

```
1 procedure void pagerank(in graph G,
2   out property<vertex(G), double> rank
3 ) {
4   long N;
5   double diff;
6   property<vertex(G), double> newRank;
7   double x;
8   N = numVertices(G);
9   x = 0.15d/(double) N;
10  setProperty(G, rank, 1d/(double) N);
11  do {
12    diff = 0.0d;
13    computeRank(G, rank, x, newRank, diff);
14    copyProperty(G, rank, newRank);
15  } while (diff > 0.001d);
16 }
17
18 iterator computeRank(
19   in graph G, in property<vertex(G), double>
       rank,
20   in double x,
21   out property<vertex(G), double> newRank,
22   in-out double diff
23 ) {
24     foreach(v: V(G)) (true) {
25       double inSum =
26         sum(w -[e]-> _: E(G, v)) (true) {
27           let
28             double rankVal = w.rank;
29             long degree = outDegree(G, w);
30           in
31             rank/(double)degree
32         };
33       double newRankVal = x + 0.85d*inSum;
34       double rankVal = v.rank;
35       diff += |newRankVal-rankVal|;
36       v.newRank = newRankVal;
37   }
38 }
```

Figure 6. PageRank algorithm with top-level iterations.

explicit semantic information. This is to ensure direct access to this information throughout the different transformations the program undergoes.

The PageRank variant in Figure 7 illustrates this normal form. All variable references are preceded with an annotation indicating whether they refer to a parameter or a local variable, and providing their type. Calls to built-in procedures are annotated with their signatures.

The transformation to establish this normal form rewrites implicitly typed expressions and references to explicitly typed counterparts, using information provided by semantic analysis.

```
procedure void pagerank(in graph G,
  out property<vertex(G), double> rank
) {
  long N;
  double diff;
  property<vertex(G), double> newRank;
  double x;
  N = [builtin (in graph)->void]numVertices(G);
  x = 0.15d/(double)[local long]N;
  [builtin(
    in graph, out property<vertex(G), double>,
    in param double)->void]
  setProperty(G, rank, 1d/(double) N);
  do {
    diff = 0.0d;
    foreach(v: V(G)) (true) {
      double inSum =
        sum(w -[e]-> _: E(G, v)) (true) {
          let
            double rankVal =
              [out
              param property<vertex(G), double>]
              w.rank;
            long deg =
              [builtin (
                in graph, in vertex(G))->void]
              outDegree(G, w);
          in
            [local double]rank
              /(double)[local long]deg
        };
      double newRankVal =
        [local double]x+ 0.85d*[local double]
    inSum;
      double rankVal =
       [out param property<vertex(G), double>]v.
    rank;
      [local double]diff +=|
        [local double]newRankVal-[local double]
    rankVal
      |;
      [out param property<vertex(G), double>]
      v.newRank =
        [local double]newRankVal;
    }
    [builtin(
      in graph, out property<vertex(G), double>,
      in  property<vertex(G), double>)->void]
    copyProperty(G, rank, newRank);
  } while ([local double]diff > 0.001d);
}
```

Figure 7. PageRank algorithm with explicit semantic information.

```
1  procedure void pagerank(in graph G,
2    out property<vertex(G), double> rank
3  ) {
4    long N;
5    double diff;
6    property<vertex(G), double> newRank;
7    double x;
8    N = numVertices(G);
9    x = 0.15d/(double) N;
10   setProperty(G, rank, 1d/(double) N);
11   do {
12     diff = 0.0d;
13     foreach[Vt: VT(G)](v: V(Vt)) (true) {
14       double inSum =
15         sum[S_w -[Ew]-> _: ET(G, Vt)]
16           (w -[e]-> _: E(Ew, v)) (true) {
17             let
18               double rankVal = w.rank;
19               long degree = outDegree(G, w);
20             in
21               rank/(double)degree
22           };
23       double newRankVal = x + 0.85d*inSum;
24       double rankVal = v.rank;
25       diff += |newRankVal-rankVal|;
26       v.newRank = newRankVal;
27     }
28     copyProperty(G, rank, newRank);
29   } while (diff > 0.001d);
30 }
```

Figure 8. PageRank algorithm with table iterators.

Explicit Table Iterators. This normal form requires explicit iterations over vertex and edge tables. This is helpful for target platforms where the runtime requires explicit iteration over tables, namely the shared-memory runtime and the database runtime.

This is shown in the PageRank variant in Figure 8, where iteration over graph tables is made explicit at lines 13 and 15. The iterator at line 13, for example, iterates over vertex tables `Vt` of the graph `G`, and over vertices `v` of every vertex table `Vt`.

The transformation to establish this normal form introduces for every element iterator its corresponding table iterator, and replaces the graph parameter in the element iterator with the table it's meant to iterate over.

Explicit Memory Management. In this normal form, properties and collections are explicitly allocated and de-allocated. This simplifies implementation, since the backends would otherwise need to analyze the entirety of the program in order to generate memory management code.

Figure 9 shows the PageRank algorithm written in this normal form. Memory is allocated for the new rank temporary property at line 7, and then de-allocated at line 30.

```
procedure void pagerank(in graph G,
  out property<vertex(G), double> rank
) {
  long N;
  double diff;
  property<vertex(G), double> newRank;
  allocate rank;
  allocate newRank;
  double x;
  N = numVertices(G);
  x = 0.15d/(double) N;
  setProperty(G, rank, 1d/(double) N);
  do {
    diff = 0.0d;
    foreach(v: V(G)) (true) {
      double inSum =
        sum(w -[e]-> _: E(G, v)) (true) {
          let
            double rankVal = w.rank;
            long degree = outDegree(G, w);
          in
            rank/(double)degree
        };
      double newRankVal = x + 0.85d*inSum;
      double rankVal = v.rank;
      diff += |newRankVal-rankVal|;
      v.newRank = newRankVal;
    }
    copyProperty(G, rank, newRank);
  } while (diff > 0.001d);
  free rank;
  free newRank;
}
```

Figure 9. PageRank algorithm with explicit memory management.

The transformation to establish this normal form generates explicit memory allocation and de-allocation instructions for all declarations of properties and collections.

Privatization. In this normal form, iterations and aggregations are surrounded with statements for initializing and finalizing thread-private variables. This is required by backends that distribute iteration workload over multiple threads or, in the case of the distributed backend, multiple machines.

This is shown in the PageRank variant in Figure 10. Line 44 creates a thread-private variable, `diffPrv`, which is then passed to the iteration to store the result of the aggregation. Line 46 reduces `diffPrv` to the original aggregation variable.

The transformation to establish this normal form creates a thread-private variable for each parallel iteration performing a reduction. Variable reductions inside the iteration are replaced by reductions to this newly-created variable. The thread-private variable is reduced to the original reduction

```
procedure void pagerank(in graph G,
  out property<vertex(G), double> rank
) {
  long N;
  double diff;
  property<vertex(G), double> newRank;
  double x;
  N = numVertices(G);
  x = 0.15d/(double) N;
  setProperty(G, rank, 1d/(double) N);
  do {
    diff = 0.0d;
    execComputeRank(G, rank, x, diff, newRank);
    copyProperty(G, rank, newRank);
  } while (diff > 0.001d);
}

iterator computeRank(
  in graph G,
  in property<vertex(G), double> rank,
  in double x, in-out double diff,
  out property<vertex(G), double> newRank
) {
  foreach(v: V(G)) (true) {
      double inSum =
        sum(w -[e]-> _: E(G, v)) (true) {
          let
            double rankVal = w.rank;
            long degree = outDegree(G, w);
          in
            rank/(double)degree
        };
      double newRankVal = x + 0.85d*inSum;
      double rankVal = v.rank;
      diff += |newRankVal-rankVal|;
      v.newRank = newRankVal;
    }
  }
}

procedure void execComputeRank(
  in graph G,
  in property<vertex(G), double> rank,
  in double x, in-out double diff,
  out property<vertex(G), double> newRank
) {
  double diffPrv = 0.0d;
  computeRank(G, rank, x, diffPrv, newRank);
  diff += diffPrv;
}
```

Figure 10. PageRank algorithm with privatization statements.

Table 2. Transformations implemented over the IR.

	Single-issue iterations	Top-level iterations	Explicit semantic information	Explicit table iterators	Explicit memory management	Privatization
Split iterations	●	○	○	○	○	○
Merge iterations	×	○	○	○	○	○
Extract iterations to top-level	○	●	○	○	○	○
Inline iterations	×	○	○	○	○	○
Add heterogeneous iterations	○	○	○	●	○	○
Explicate types	○	○	●	○	○	○
Explicate memory management	○	○	○	○	●	○
Privatise iterations	○	○	○	○	○	●
Lower to query IR	○	○	×	○	○	○
Eliminate dead code	○	○	○	○	○	○
Add initializers	○	○	○	○	○	○
Precompute degree property	○	○	○	○	○	○
Common neighbor iterations	○	○	○	○	○	○
Multi-source BFS	○	○	○	○	○	○

variable after the iteration completes. This way, each thread performing a chunk of the iteration will initialize the thread-private variable, and use it to store its computation. These computations will then be reduced across threads to give the total reduction value.

Transformations and Normal Forms. Normal forms act as prerequisites for transformations. These transformations are then composed into a multi-stage transformation. This way, we break down the semantic gap towards the target platforms into several smaller gaps bridged by these transformations [40]. We distinguish three kinds of transformations with respect to normal forms. A transformation t over language L *establishes* a normal form N, iff it is applicable to all programs of L and always yields programs in N. That is, the transformation is a total function $t : L \to N$. A transformation t *preserves* a normal form N, iff it is applicable to all programs of N (and potentially more) and yields programs in N for inputs in N. That is, the transformation restricted to programs in N is a total function $t|_N : N \to N$. Otherwise, t *breaks* the normal form N.

Table 2 indicates for each transformation we implement, whether it establishes (●), breaks (×), or preserves (○) our normal forms. Some of these transformations are generic, and can be applied regardless of the targeted backend, such as the transformations to eliminate dead code, or to add variable initializers. Others are more particular, but still common between all four of our backends, such as the transformations to extract iterations at top level, or explicate type information. This leaves transformations which are specific to particular backends, such as adding table iterators, or splitting iterations, both of which are required by the relational database backend. We also implement domain-specific optimizations, such as the common neighbor iterator (CNI) optimization which identifies common neighbors for pairs of vertices in a graph [30], the optimization to precompute the degree of each vertex and store it in a property depending on a heuristic, or the multi-source BFS (MS-BFS) optimization which performs several BFS traversals at the same time, starting from different roots [29, 33].

4 Implementation

We implement the *PGX Algorithm* Compiler in Spoofax, a language workbench for developing textual DSLs [15, 44]. In this section, we report on our experience using Spoofax and highlight some of the challenges and requirements that modern language workbenches [8] need to meet in order to be suitable for industrial use.

Compiler Components. Figure 1 gives an overview of the components of the *PGX Algorithm* compiler. The IR component is at the center of our implementation. This component provides static analyses, normal forms, and transformations on the intermediate representation. Frontend components provide parsers for different DSLs and map programs to the intermediate representation. We currently provide two different frontends for *Green-Marl* and *PGX Algorithm*. *Green-Marl* [11] is a DSL for algorithmic graph processing. The frontend maps its syntax to the intermediate representation. *PGX Algorithm* is an internal DSL with Java syntax, providing similar language concepts as *Green-Marl*. The *PGX Algorithm* frontend integrates with the *javac* compiler and maps Java constructs to the intermediate representation. Backend components generate code for the various target platforms. We currently support backends for a single-machine runtime, a distributed runtime, an in-memory database and a relational

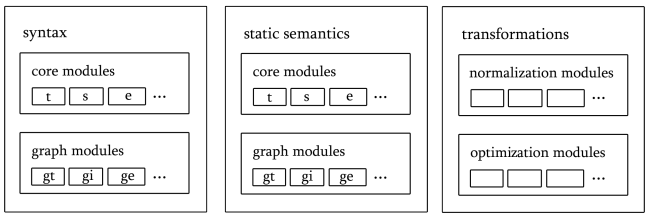

Figure 11. Module organization of the IR.

database. The generated code is represented as abstract syntax trees of the target language. Pretty-printing components turn these abstract syntax trees into program code.

Support for component-based compiler implementations is crucial for industrial use cases. It allows various parties to extend the compiler with additional components. New frontend components can map other graph processing DSLs to the intermediate representation. New backend components can be added to support additional target platforms. In this context, the ability to combine open-source and proprietary components is important. In the current implementation, we rely on an existing open-source component from the Spoofax ecosystem to pretty-print Java code, but maintain our own pretty-printing components for C/C++ and PL/SQL code. The component-based approach allows us to switch to open-source components when they become available. We also have the option to release the IR component as an open-source project, without exposing the entirety of the compiler, including proprietary optimizations or backend components. This would allow academic and industrial partners to target the IR from other DSLs, to experiment with new optimizations, or to provide new backends targeting additional execution platforms. Finally, a component-based implementation helps to compose different variants of the compiler. This is important if the compiler is bundled in multiple products with different execution platforms. For example, several of our products ship with the single-machine and distributed runtimes, and a variant of the *PGX Algorithm* compiler which only supports code generation for these runtimes.

We implemented each compiler component as a separate Spoofax project. Spoofax supports both source code and runtime dependencies between components. We found the handling of source code dependencies to be problematic, since the same code will end up in multiple components at runtime, increasing the memory footprint of the compiler at runtime. However, this will be solved in future releases of Spoofax which rely on flexible, composable compiler pipelines [17, 18].

Modular Language Design. We organize the implementation of each compiler component in hierarchical modules. Spoofax provides support for such a modular implementation in all of its meta-languages [16]. Figure 11 shows the modules of the IR component. Syntax modules define concrete and abstract syntax for the intermediate representation in SDF3 [7]. Spoofax' support for language composition allows us to use concrete syntax in transformations [38] and tests [13]. Static semantic modules define name binding and typing rules in Statix [37]. Finally, transformation modules implement various transformations over the intermediate representation [3, 39]. At the syntax level, we distinguish between a core of imperative language constructs, and domain-specific constructs for graph processing, which are defined in separate submodules. This distinction is mirrored in the static semantic modules and in the backend components, which separate transformations to generate code for imperative and domain-specific constructs into different modules.

We found the hierarchical modularization of our implementation extremely helpful when developing, extending, and maintaining the compiler. In the initial development, it allowed us to first focus on the simpler core constructs, before we added domain-specific constructs step-by-step. We follow the same approach when extending the implementation with either new language constructs or new backend components. In maintenance, the module system helps us to quickly locate the relevant pieces of the implementation for a specific language construct, and to adapt it with a number of typically local changes. It also helped new team members to easily navigate the code base and explore it systematically from the more easy core constructs to the more complicated domain-specific constructs. A modular implementation can also reduce build times of a compiler project significantly, when the language workbench supports incremental language builds [31].

Product Integration. The *PGX Algorithm* compiler is integrated into several industrial products, which allows customers to compile and run graph processing algorithms as part of their graph analytics workflows. These products provide an API to compile and run algorithms. Implementing such an API requires the ability to programmatically compose and invoke compilation pipelines. While many language workbenches pride themselves on providing DSL editors which are integrated into modern IDEs [8], it is important to also support use cases where compilers can be used without editors and outside IDEs. Spoofax currently provides an API for these use cases. Future versions of Spoofax will allow to specify flexible, composable compiler pipelines in a DSL [17, 18].

The deep integration of a compiler into the code of an industrial product puts extra requirements on the code of the compiler and, by extension, the language workbench runtimes, which are required to run the compiler. For example, the product might become vulnerable to security exploits in such a runtime or in its dependencies. To mitigate such risks, language workbench development teams need to establish processes to monitor and fix vulnerabilities in dependencies.

This typically leads to frequent version updates for dependencies and frequent releases of the language workbench itself, to address vulnerabilities. Software licenses for dependencies affect not only the language workbench runtime, but also the compiler relying on that runtime and the product integrating the compiler. This might inhibit the use of a language workbench for some industrial use cases, if a software license for one of its runtime dependencies is not acceptable.

The integration of the *PGX Algorithm* compiler into products extends to testing. We need to test the compiler, its target runtimes, and their integration in the product. As compilers and runtimes tend to evolve in parallel, we must continuously ensure that the code generated by an evolved compiler behaves correctly on an evolved runtime. Language workbenches need to support testing of the various compiler components, of compilation pipelines composed of these components, and of code integrating compilers into larger products. We implement a separate testing component for each compiler component. Each testing component comprises declarative tests written in Spoofax' SPT language [14]. These tests can be run programmatically and are integrated into our continuous integration pipelines. We have additional tests verifying that the compiler can generate code, that the generated code can be compiled for the target platform, and that the compiled algorithm yields the correct results when executed on the target platform. These tests are also integrated into our continuous integration pipelines. Build times for compiler components as well as execution times for compiler-related tests have a significant influence on the execution times of continuous integration pipelines. This can become more important than the performance of a compiler as part of the product, when typically only a few algorithms are compiled.

5 Related Work

DSLs for Graph Analytics. We can distinguish two kinds of workloads when analyzing graph data. *Graph queries* aim to match patterns as subgraphs of the analyzed graph. *Graph algorithms* perform complex traversals over a graph to explore the paths between vertices, the importance of vertices, or the clustering of vertices. There exists a variety of DSLs to support these workloads.

Neo4j's *Cypher* [23] and Oracle's *PGQL* [24] are prominent examples of query languages for property graphs. Both DSLs allow for graph pattern matching through SQL-like queries. Apache's *Gremlin* query language takes a different approach by providing means to express graph traversals as a sequence of atomic operation on a data stream. This allows to implement pattern-matching queries and simple graph algorithms such as basic centrality algorithms. It is not well-suited to express more complex algorithms such as PageRank.

Falcon [5, 6] and *GraphIt* [45] are examples of languages for implementing complex graph algorithms. *Falcon* extends the C language with a set of high-level graph-specific constructs, abstracting over hardware-related implementation details. It requires algorithms to be explicitly parallel, and while most constructs are hardware-independent, some annotation constructs depend on the underlying platform. This requires users to provide slightly different implementations depending on the target platform. *Falcon* supports algorithms which can change the structure of the input graph, which is currently not supported in our IR. *GraphIt* provides a language for expressing the logic of graph algorithms, and a separate scheduling language for specifying performance optimizations. This scheduling language adds support for a number of functions through which the user can fine-tune their algorithm's performance, by composing optimizations for edge traversal direction, vertex data layout, and program structure. This way, users can navigate the tradeoff space between locality, parallelism, and work-efficiency for faster algorithm implementations manually. In contrast, our work aims at compilers picking optimizations.

Intermediate Representations. There has been a large amount of work conducted in the way of creating well-developed common IRs for compiler architectures. LLVM [19] provides a low-level, assembly-like IR for a unified code representation of high-level languages. This makes it unsuitable for our usecase, as our target platforms require higher-level paradigms than what LLVM offers.

Many compilers utilize the LLVM toolchain such as not to reinvent the wheel, but implement their own high-level IRs on top of it in order to bridge the gap between the source language and the low-level abstractions of LLVM IR. The *Swift* compiler [1] and the *Rust* compiler [21] are two such compilers. *Swift* and *Rust* are both multi-paradigm, general-purpose compiled languages. The *Swift* compiler translates source code to a high-level intermediate language – the *Swift* Intermediate Language (SIL) [36] – before applying a series of optimization passes and diagnostic passes, and lowering code to LLVM IR. The *Rust* compiler also introduces a middle IR (MIR) [22] that's essentially a simplified version of the *Rust* language: source code is translated into a high-level IR (HIR) and then lowered into MIR at a later stage, thus simplifying compilation and enabling high-level optimizations.

MLIR, another project under the LLVM umbrella, provides a customizable IR that allows its users to define the right level of abstraction for their programs [20]. Sujeeth et al. [32] show how to compose DSLs of different domains embedded together in a single program using *Delite*, a compiler framework for building parallel DSLs, targeting a parallel runtime with a heterogeneous architecture underneath [4]. *Delite* DSLs are embedded in Scala, and then transformed into a common IR.

DSL compilers may still resort to developing their own IRs in order to solve domain-specific problems. Pai et al. present a compiler which generates CUDA code for programs written in IrGL, an intermediate representation for graph algorithms [27]. This architecture, however, does not share the multi-target, multi-paradigm characteristic of our compiler. The *Falcon* compiler targets single machine and distributed systems with different architectures (multicore CPUs, single GPUs, multi-GPUs, and heterogeneous backends), with a platform independent IR at the center of its compiler [5, 6]. To our knowledge, it does not target a database environment. The *Unified GraphIt Compiler* (UGC) implements a graph-specific IR targeting four backends (*GraphVMs*) with different hardware architectures and, consequently, different paradigms [2]. While UGC also targets platforms with diverse architectures, this diversity is at a hardware-level, thus a lower level than what we deal with in the scope of our compiler.

DSL Modularization. Ratiu et al. show how layered modularization of languages enables an easy, efficient, and reusable implementation of domain-specific analyses [28]. This efficiency motivates our design of a modular architecture for the *PGX Algorithm* compiler and its IR. *mbeddr* is well-known for its modular language implementation. *mbeddr* is a set of modular and incremental language extensions for embedded software development, based on C [41]. Programs in *mbeddr* are heterogeneous in nature, since they can introduce different languages (e.g. a C program with embedded state machines). This makes *mbeddr* well-suited for multi-paradigm programming. *mbeddr*'s language extensions are desugared into plain C code. In our compiler, the domain-specific extension is not desugared; instead, it is translated by each backend into suitable code in the target language.

mbeddr is built using JetBrains MPS, a projectional language workbench. MPS' projectional nature enhances modularization of implementation, and allows it to support simple extension, embedding and composition of unrelated languages [42]. We found modular implementations to be equally well supported by Spoofax.

6 Conclusions and Future Work

We have presented a compiler architecture centered around a syntactically rich IR for algorithmic graph processing as well as transformations and optimizations over this IR, which compiler backends can share and compose as needed. The presented solution allows us to efficiently develop and maintain a compiler for *PGX Algorithm*, a DSL for algorithmic graph processing, with multiple backends targeting different execution runtimes and programming paradigms. To conclude this paper, we discuss several aspects of our solution, with a particular focus on directions for future work.

General Applicability. Our approach is currently limited to a concrete industrial use case. We plan to explore several directions to generalize this approach in the future. First, our IR should be a suitable target for other DSLs for algorithmic graph processing, but might not be as suitable for more query-like data processing patterns. We plan to investigate a more general IR for various data processing workloads, including machine learning workloads. Second, we are working on extending the set of domain-specific‘ optimizations. This involves studying the applicability of new optimizations, their interactions, and their impact on normal forms.

Formal Semantics. We have introduced several normal forms, transformations, and optimizations over our IR. So far, these are of strictly practical use in the implementation of the *PGX Algorithm* compiler. We test this implementation extensively using Spoofax' testing capabilities [14]. In a next step, formal specifications of the dynamic semantics of the IR as well as transformations and optimizations are needed to reason more formally about the correctness and composability of transformations. This will also allow us to reason about the relationships of normal forms and their compatibility.

Performance of Graph Algorithms. Oracle ships graph algorithms as part of several products with graph-processing capabilities. These algorithms are implemented in *PGX Algorithm* and compiled to the target platform of the product. The performance of the algorithms is influenced by the effectiveness of the optimizations applied by the compiler, the compilation pattern of the backend, and the performance of the target platform itself.

We previously evaluated the impact of compiler optimizations on performance in the single-machine runtime [29]. We showed that graph algorithms implemented in *Green-Marl* outperform implementations in general-purpose languages with API calls into the runtime. For example, the *Green-Marl* implementation of PageRank takes less than a second to compile and to calculate ranks for LiveJournal [9], a graph just short of 5 million vertices and 69 million edges. An API-based implementation is two orders of magnitude slower. In the same study, we also evaluated the impact of compiler optimizations on several graph algorithms. For PageRank, the optimized implementation was 1.5 times faster on LiveJournal, and 2 times faster on a Twitter graph with over a billion edges. In a separate study, we compared the performance of the distributed runtime against state-of-the-art distributed graph processing engines based on execution times of a set of widely-used graph algorithms [12]. The study showed that the compiled algorithms outperform algorithms running on other engines by one to two orders of magnitude.

We see similar performance on these target platforms with our new compiler. We currently cannot disclose performance of algorithms in the database runtimes, since these runtimes are still in an early experimental stage.

References

[1] Apple. 2022. *The Swift Compiler.* Retrieved April 04, 2022 from https://www.swift.org/swift-compiler/

[2] Ajay Brahmakshatriya, Emily Furst, Victor A. Ying, Claire Hsu, Changwan Hong, Max Ruttenberg, Yunming Zhang, Dai Cheol Jung, Dustin Richmond, Michael B. Taylor, Julian Shun, Mark Oskin, Daniel Sanchez, and Saman Amarasinghe. 2021. Taming the Zoo: The Unified GraphIt Compiler Framework for Novel Architectures. In *Proceedings of the 48th Annual International Symposium in Computer Architecture (ISCA).*

[3] Martin Bravenboer, Arthur van Dam, Karina Olmos, and Eelco Visser. 2006. Program Transformation with Scoped Dynamic Rewrite Rules. *Fundamenta Informaticae* 69, 1-2 (2006), 123–178. https://content.iospress.com/articles/fundamenta-informaticae/fi69-1-2-06

[4] Hassan Chafi, Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Anand R. Atreya, and Kunle Olukotun. 2011. A Domain-Specific Approach to Heterogeneous Parallelism. *SIGPLAN Not.* 46, 8 (Feb. 2011), 35–46. https://doi.org/10.1145/2038037.1941561

[5] Unnikrishnan Cheramangalath, Rupesh Nasre, and Y. N. Srikant. 2015. Falcon: A Graph Manipulation Language for Heterogeneous Systems. 12, 4, Article 54 (2015), 27 pages. https://doi.org/10.1145/2842618

[6] Unnikrishnan Cheramangalath, Rupesh Nasre, and Y. N. Srikant. 2020. *Falcon: A Domain Specific Language for Graph Analytics.* Springer International Publishing, Cham, 153–179. https://doi.org/10.1007/978-3-030-41886-1_7

[7] Luis Eduardo de Souza Amorim and Eelco Visser. 2020. Multi-purpose Syntax Definition with SDF3. In *Software Engineering and Formal Methods - 18th International Conference, SEFM 2020, Amsterdam, The Netherlands, September 14-18, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12310)*, Frank S. de Boer and Antonio Cerone (Eds.). Springer, 1–23. https://doi.org/10.1007/978-3-030-58768-0_1

[8] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël D. P. Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido H. Wachsmuth, and Jimi van der Woning. 2013. The State of the Art in Language Workbenches. In *Software Language Engineering*, Martin Erwig, Richard F. Paige, and Eric Van Wyk (Eds.). Springer International Publishing, Cham, 197–217.

[9] Jennifer Golbeck and Matthew Rothstein. 2008. Linking Social Networks on the Web with FOAF: A Semantic Web Case Study., Vol. 2. 1138–1143.

[10] T. Harris and S. Kaestle. 2015. Callisto-RTS: Fine-Grain Parallel Loops. In *USENIX Annual Technical Conference.*

[11] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. 2012. Green-Marl: A DSL for Easy and Efficient Graph Analysis. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems* (London, England, UK) *(ASPLOS XVII).* Association for Computing Machinery, New York, NY, USA, 349–362. https://doi.org/10.1145/2150976.2151013

[12] Sungpack Hong, Siegfried Depner, Thomas Manhardt, Jan Van Der Lugt, Merijn Verstraaten, and Hassan Chafi. 2015. PGX.D: A Fast Distributed Graph Processing Engine. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Austin, Texas) *(SC '15).* Association for Computing Machinery, New York, NY, USA, Article 58, 12 pages. https://doi.org/10.1145/2807591.2807620

[13] Lennart C.L. Kats, Rob Vermaas, and Eelco Visser. 2011. Integrated Language Definition Testing: Enabling Test-Driven Language Development. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Portland, Oregon, USA) *(OOPSLA '11).* Association for Computing Machinery, New York, NY, USA, 139–154. https://doi.org/10.1145/2048066.2048080

[14] Lennart C.L. Kats, Rob Vermaas, and Eelco Visser. 2011. Testing Domain-Specific Languages. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion* (Portland, Oregon, USA) *(OOPSLA '11).* Association for Computing Machinery, New York, NY, USA, 25–26. https://doi.org/10.1145/2048147.2048160

[15] Lennart C.L. Kats and Eelco Visser. 2010. The Spoofax Language Workbench. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion* (Reno/Tahoe, Nevada, USA) *(OOPSLA '10).* Association for Computing Machinery, New York, NY, USA, 237–238. https://doi.org/10.1145/1869542.1869592

[16] Lennart C.L. Kats, Eelco Visser, and Guido Wachsmuth. 2010. Pure and Declarative Syntax Definition: Paradise Lost and Regained. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Reno/Tahoe, Nevada, USA) *(OOPSLA '10).* Association for Computing Machinery, New York, NY, USA, 918–932. https://doi.org/10.1145/1869459.1869535

[17] Gabriël Konat, Sebastian Erdweg, and Eelco Visser. 2018. Scalable incremental building with dynamic task dependencies. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, Marianne Huchard, Christian Kästner, and Gordon Fraser (Eds.). ACM, 76–86. https://doi.org/10.1145/3238147.3238196

[18] Gabriël Konat, Michael J. Steindorfer, Sebastian Erdweg, and Eelco Visser. 2018. PIE: A Domain-Specific Language for Interactive Software Development Pipelines. *The Art, Science, and Engineering of Programming* 2, 3 (Mar 2018). https://doi.org/10.22152/programming-journal.org/2018/2/9

[19] C. Lattner and V. Adve. 2004. LLVM: a compilation framework for lifelong program analysis amp; transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 75–86. https://doi.org/10.1109/CGO.2004.1281665

[20] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO).* 2–14. https://doi.org/10.1109/CGO51591.2021.9370308

[21] Heng Lin, Piyo Chen, Yuan-Shin Hwang, and Jenq-Kuen Lee. 2019. Devise Rust Compiler Optimizations on RISC-V Architectures with SIMD Instructions. In *Proceedings of the 48th International Conference on Parallel Processing: Workshops* (Kyoto, Japan) *(ICPP 2019).* Association for Computing Machinery, New York, NY, USA, Article 13, 7 pages. https://doi.org/10.1145/3339186.3339193

[22] Niko Matsakis. 2016. *Introducing MIR.* Retrieved April 04, 2022 from https://blog.rust-lang.org/2016/04/19/MIR.html

[23] Neo4j. 2022. *Cypher Query Language.* Retrieved April 04, 2022 from https://neo4j.com/developer/cypher/

[24] Oracle. 2022. *PGQL Property Graph Query Language.* Retrieved April 04, 2022 from https://pgql-lang.org

[25] Oracle. 2022. *PGX documentation.* Retrieved April 04, 2022 from https://docs.oracle.com/cd/E56133_01/latest/reference/analytics/pgx-algorithm.html

[26] Lawrence Page. 2001. Method for node ranking in a linked database. US Patent No. 6,285,999 B1, Filed Jan. 9th, 1998; Issued Sep. 4, 2001.

[27] Sreepathi Pai and Keshav Pingali. 2016. A Compiler for Throughput Optimization of Graph Algorithms on GPUs. *SIGPLAN Not.* 51, 10 (Oct. 2016), 1–19. https://doi.org/10.1145/3022671.2984015

[28] Daniel Ratiu, Markus Voelter, Zaur Molotnikov, and Bernhard Schaetz. 2012. Implementing Modular Domain Specific Languages and Analyses. In *Proceedings of the Workshop on Model-Driven Engineering, Verification and Validation* (Innsbruck, Austria) *(MoDeVVa '12).* Association for Computing Machinery, New York, NY, USA, 35–40. https://doi.org/10.1145/2427376.2427383

[29] Martin Sevenich, Sungpack Hong, Oskar van Rest, Zhe Wu, Jayanta Banerjee, and Hassan Chafi. 2016. Using Domain-Specific Languages for Analytic Graph Databases. *Proc. VLDB Endow.* 9, 13 (Sept. 2016), 1257–1268. https://doi.org/10.14778/3007263.3007265

[30] Martin Sevenich, Sungpack Hong, Adam Welc, and Hassan Chafi. 2014. Fast In-Memory Triangle Listing for Large Real-World Graphs. In *Proceedings of the 8th Workshop on Social Network Mining and Analysis* (New York, NY, USA) *(SNAKDD'14)*. Association for Computing Machinery, New York, NY, USA, Article 2, 9 pages. https://doi.org/10.1145/2659480.2659494

[31] Jeff Smits, Gabriël Konat, and Eelco Visser. 2020. Constructing Hybrid Incremental Compilers for Cross-Module Extensibility with an Internal Build System. *Programming Journal* 4, 3 (2020), 16. https://doi.org/10.22152/programming-journal.org/2020/4/16

[32] Arvind K. Sujeeth, Tiark Rompf, Kevin J. Brown, HyoukJoong Lee, Hassan Chafi, Victoria Popic, Michael Wu, Aleksandar Prokopec, Vojin Jovanovic, Martin Odersky, and Kunle Olukotun. 2013. Composition and Reuse with Compiled Domain-Specific Languages. In *Proceedings of the 27th European Conference on Object-Oriented Programming* (Montpellier, France) *(ECOOP'13)*. Springer-Verlag, Berlin, Heidelberg, 52–78. https://doi.org/10.1007/978-3-642-39038-8_3

[33] Manuel Then, Moritz Kaufmann, Fernando Chirigati, Tuan-Anh Hoang-Vu, Kien Pham, Alfons Kemper, Thomas Neumann, and Huy T. Vo. 2014. The More the Merrier: Efficient Multi-Source Graph Traversal. *Proc. VLDB Endow.* 8, 4 (Dec. 2014), 449–460. https://doi.org/10.14778/2735496.2735507

[34] Yuanyuan Tian, En Liang Xu, Wei Zhao, Mir Hamid Pirahesh, Sui Jun Tong, Wen Sun, Thomas Kolanko, Md. Shahidul Haque Apu, and Huijuan Peng. 2020. IBM Db2 Graph: Supporting Synergistic and Retrofittable Graph Queries Inside IBM Db2. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 345–359. https://doi.org/10.1145/3318464.3386138

[35] Tinkerpop. 2022. *Tinkerpop, Gremlin.* Retrieved April 04, 2022 from https://github.com/tinkerpop/gremlin/wiki

[36] David Ungar, David Grove, and Hubertus Franke. 2017. Dynamic Atomicity: Optimizing Swift Memory Management. 52, 11 (oct 2017), 15–26. https://doi.org/10.1145/3170472.3133843

[37] Hendrik van Antwerpen, Casper Bach Poulsen, Arjen Rouvoet, and Eelco Visser. 2018. Scopes as Types. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 114 (Oct. 2018), 30 pages. https://doi.org/10.1145/3276484

[38] Eelco Visser. 2002. Meta-programming with Concrete Object Syntax. In *Generative Programming and Component Engineering, ACM SIGPLAN/SIGSOFT Conference, GPCE 2002, Pittsburgh, PA, USA, October 6-8, 2002, Proceedings (Lecture Notes in Computer Science, Vol. 2487)*, Don S. Batory, Charles Consel, and Walid Taha (Eds.). Springer, 299–315. https://doi.org/10.1007/3-540-45821-2_19

[39] Eelco Visser, Zine-El-Abidine Benaissa, and Andrew P. Tolmach. 1998. Building Program Optimizers with Rewriting Strategies. In *Proceedings of the third ACM SIGPLAN international conference on Functional programming*, Matthias Felleisen, Paul Hudak, and Christian Queinnec (Eds.). ACM, Baltimore, Maryland, United States, 13–26. https://doi.org/10.1145/289423.289425

[40] Markus Voelter. 2013. *DSL Engineering: Designing, Implementing and Using Domain-Specific Languages.* CreateSpace Independent Publishing Platform.

[41] Markus Voelter, Daniel Ratiu, Bernhard Schaetz, and Bernd Kolb. 2012. Mbeddr: An Extensible C-Based Programming Language and IDE for Embedded Systems. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity* (Tucson, Arizona, USA) *(SPLASH '12)*. Association for Computing Machinery, New York, NY, USA, 121–140. https://doi.org/10.1145/2384716.2384767

[42] Markus Völter and Eelco Visser. 2010. Language Extension and Composition with Language Workbenches. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion* (Reno/Tahoe, Nevada, USA) *(OOPSLA '10)*. Association for Computing Machinery, New York, NY, USA, 301–304. https://doi.org/10.1145/1869542.1869623

[43] W3C. 2022. *SPARQL Query Language for RDF.* Retrieved April 04, 2022 from https://www.w3.org/TR/rdf-sparql-query/

[44] Guido H. Wachsmuth, Gabriël D.P. Konat, and Eelco Visser. 2014. Language Design with the Spoofax Language Workbench. *IEEE Software* 31, 5 (2014), 35–43. https://doi.org/10.1109/MS.2014.100

[45] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. 2018. GraphIt: A High-Performance Graph DSL. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 121 (Oct. 2018), 30 pages. https://doi.org/10.1145/3276491