Spoofax at Oracle: Domain-Specific Language Engineering for Large-Scale Graph Analytics

Houda Boukham
Mohammed V University in Rabat,
Ecole Mohammadia d'Ingénieurs, Morocco
Oracle Labs, Casablanca, Morocco

Guido Wachsmuth
Oracle Labs, Zürich, Switzerland

Toine Hartman
Oracle Labs, Utrecht, The Netherlands

Hamza Boucherit
Oracle Labs, Casablanca, Morocco

Oskar van Rest
Oracle, Redwood Shores, CA, USA

Hassan Chafi
Oracle Labs, Zürich, Switzerland

Sungpack Hong
Oracle Labs, Redwood Shores, CA, USA

Martijn Dwars
Oracle Labs, Zürich, Switzerland

Arnaud Delamare
Oracle Labs, Zürich, Switzerland

Dalila Chiadmi
Mohammed V University in Rabat,
Ecole Mohammadia d'Ingénieurs, Morocco

Abstract

For the last decade, teams at Oracle relied on the *Spoofax* language workbench to develop a family of domain-specific languages for graph analytics in research projects and in product development. In this paper, we analyze the requirements for integrating language processors into large-scale graph analytics toolkits and for the development of these language processors as part of a larger product development process. We discuss how *Spoofax* helps to meet these requirements and point out the need for future improvements.

2012 ACM Subject Classification Software and its engineering → Domain specific languages; Software and its engineering → Translator writing systems and compiler generators

Keywords and phrases language workbench, domain-specific language

Digital Object Identifier 10.4230/OASIcs.EVCS.2023.5

Acknowledgements The following people contributed to Spoofax implementations of domain-specific languages for graph analytics as presented in this paper (in alphabetical order): Alexander Weld, Amin Ait Lamqadem, Arnaud Delamare, Calin Iorgulescu, Daco Harkes, Daniel Lehmann, Daniel Pelsmaeker, Danny Groenewegen, Davide Basilio Bartolini, Eelco Visser, Felix Kaser, Gabriel Konat, Guido Wachsmuth, Hamid El Maazouz, Hamza Boucher, Houda Boukham, Hugo Kapp, Ivo Wilms, Jan van der Lugt, Jeff Smits, Jinha Kim, Jinsoo Lee, Korbinian Schmid, Luis Eduardo de Souza Amorim, Manuel Then, Martijn Dwars, Martin Sevenich, Matthijs Bijman, Mohammed Bacha, Moritz Baumann, Mourad Abbay, Omar Heddi, Oskar van Rest, Ossama Elbourakkadi, Patrick Gruntz, Ragahavan Raman, Rob Vermaas, Siegfried Depner, Stefan Kaestle, Sungpack Hong, Toine Hartman, Vlad Vergu.

1 Introduction

In 2011, the *Parallel Graph AnalytiX* (*PGX*) team at Oracle Labs faced a challenge in its *Green-Marl* [1] compiler. The C++ implementation became increasingly difficult to maintain and slowed down the exploration of powerful optimization techniques. The *PGX* team was looking for alternatives, and identified declarative language specifications as advocated by the *Spoofax* project [2] as a promising approach to reduced maintenance efforts and to faster exploration of potential optimizations. In 2012, Oracle started a collaboration with Eelco

Visser's research group at Delft University of Technology. Eelco's group started to develop a first language definition of *Green-Marl* in *Spoofax*, consisting of an *SDF2* [11, 12] grammar, name binding rules in the new name binding language *NaBL* [4], and type specifications in *TS* [13], an emerging meta-language for type specifications. Green-Marl provided a valuable industrial case study for the work on *NaBL* and *TS*, and appeared in several publications on *Spoofax* and its metalanguages. In 2013, the *PGX* team took over the work on the frontend and started to develop *Stratego* transformation rules for a compiler backend in *Spoofax*, targeting a Java-based runtime for large scale parallel graph analytics [8].

A decade of collaboration later, teams at Oracle rely on *Spoofax* to develop a family of domain-specific languages for graph analytics in research projects and in product development:

Green-Marl is a DSL for algorithmic graph processing [1]. We use it internally to implement over 60 graph algorithms which are available in Oracle products for graph analytics.

PGX Algorithm is another DSL for algorithmic graph processing [7]. It provides the same domain-specific constructs as *Green-Marl*, but captures them in a *Java*-API. *PGX Algorithm* is part of several Oracle products for graph analytics, allowing customers to implement their own graph algorithms.

PGQL is an *SQL*-like graph query language [6, 10] and is integrated into several Oracle products.

These languages and their language processors are building blocks of larger toolkits and Oracle products for graph analytics. In this paper, we analyze the requirements for integrating language processors into large scale graph analytics toolkits (Section 2) and for the development of these language processors as part of a larger product development process (Section 3). We discuss how *Spoofax* helps to meet these requirements and point out the need for future improvements.

2 Domain-Specific Language Processors for Graph Analytics

Large-scale graph analytics solutions typically adopt a client-server architecture. Users interact with a client such as a console or a notebook, while the actual graph data is processed on a server. Domain-specific languages such as Neo4j's *Cypher* query language [5], Apache TinkerPop's *Gremlin* graph traversal language [9], or Oracle's *PGX Algorithm* and *PGQL* need to be integrated into the client-server architecture. Typically, users send domain-specific code from the client to the server, where it is processed.

▶ **Example 1** (Graph analytics with *PGX*)**.** Figure 1 provides a simple interaction, in which a user compiles and runs a custom graph algorithm written in *PGX Algorithm*, before querying the result with a *PGQL* query. Figure 2 reproduces a similar interaction on a notebook. There, the query run by the user matches triangle patterns against graph data, and visualizes the result. Graph algorithm and query are processed on the server. Figure 3 illustrates the components of the *PGX* server which are involved in this process. *Spoofax*-implemented components are highlighted in gray.

The *PGX* server passes the algorithm code to the *PGX Algorithm* compiler for parsing, static analysis, optimization, and target code generation. The generated code is then compiled and loaded into the server. Finally, the server returns a symbolic handle of the compiled algorithm to the client. The user can then run the compiled algorithm. The server executes the algorithm and returns a summary of the execution to the client.

Similarly, the client sends *PGQL* queries to the server, where they are processed and executed. The *PGQL* compiler first parses the query, extracts the name of the queried graph, and looks up its metadata. Query and metadata are then statically analyzed for

Figure 1 Running a custom graph algorithm and a *PGQL* query in the *PGX Java* shell.

```
PgxGraph G = session.readGraphWithProperties("twitter.edge.json");
==> PgxGraph[name=Twitter,N=41652230,E=1468365182]

pgx> var compiled = session.compileProgram("DegreeCentrality.java");
==> CompiledProgram[name=degreeCentrality]

pgx> compiled.run(G, degree);
==> {"success" : true, "exception" : null, "returnValue" : null}

pgx> G.queryPgql("SELECT id(v), v.degree FROM MATCH (v) ORDER BY v.degree DESC LIMIT 3").
     print()
+------------------+
| id(v)  | v.degree |
+------------------+
| 10009 | 22889    |
| 37356 | 15554    |
| 87    | 15218    |
+------------------+
```

name and type errors. Next, the compiler performs static optimizations on the query, before passing the query to the query optimizer. The query optimizer determines the most efficient execution plan of a query as a sequence of operators that can perform the necessary scanning, computation, and manipulation of graph data on the server. Finally, the result of the query is returned to the client.

Throughout the remainder of this section, we collect requirements for the integration of language processors into client-server architectures and report on our specific experiences with language processors derived from language definitions in *Spoofax*.

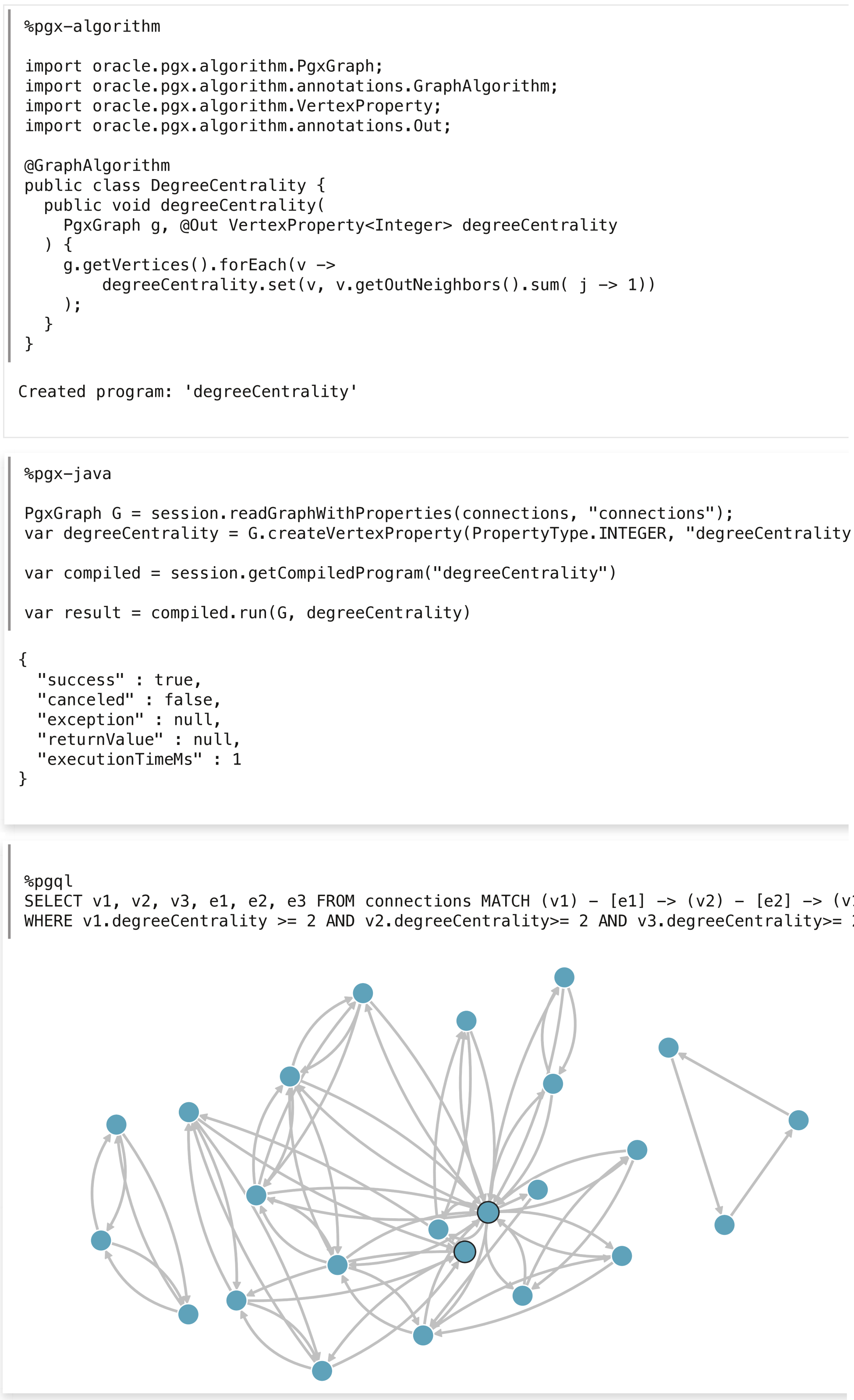
▶ **Requirement 1** (Programmatic Integration). *Language processors cannot be stand-alone tools but need to be programmatically integrated with the server.*

In its early stages, the main focus of the *Spoofax* project was to provide language processors for modern IDEs. Later, *Spoofax* also supported the generation of stand-alone language processors which could be invoked from command-line. Only *Spoofax 2* [3] introduced *Java* APIs to invoke language processors programmatically. We currently rely on this API to integrate the *PGX Algorithm* and *PGQL* compilers into the *PGX* server runtime.

▶ **Requirement 2** (Compliant Dependencies). *Language processors might depend on external software libraries. These dependencies need to comply with corporate policies for dependencies. Such policies might forbid particular versions of software libraries due to known vulnerabilities or entirely forbid the use of an external library in favor of internal solutions.*

PGX Algorithm and *PGQL* compilers depend on the *Spoofax* runtime, which itself has many dependencies on external software. Many of these dependencies are introduced to support IDE use cases, for example to load and reload language processors dynamically at runtime. This is problematic, since these dependencies are not required by our compilers, but still need to be regularly checked for compliance. The current work on *Spoofax 3* results in a notable decrease in dependencies.

▶ **Requirement 3** (Secure Language Processors). *Language processors run on the server and constitute potential targets for attacks. These can be Denial-of-Service attacks with extremely large or frequent requests to language processors. Language processors need to be embedded in a protective structure, which allows them to be interrupted when they hit a maximum processing time and to be shut down and restarted when they are in an exceptional state, for*



```
%pgx-algorithm

import oracle.pgx.algorithm.PgxGraph;
import oracle.pgx.algorithm.annotations.GraphAlgorithm;
import oracle.pgx.algorithm.VertexProperty;
import oracle.pgx.algorithm.annotations.Out;

@GraphAlgorithm
public class DegreeCentrality {
  public void degreeCentrality(
    PgxGraph g, @Out VertexProperty<Integer> degreeCentrality
  ) {
    g.getVertices().forEach(v ->
        degreeCentrality.set(v, v.getOutNeighbors().sum( j -> 1))
    );
  }
}
```

```
Created program: 'degreeCentrality'
```

```
%pgx-java

PgxGraph G = session.readGraphWithProperties(connections, "connections");
var degreeCentrality = G.createVertexProperty(PropertyType.INTEGER, "degreeCentrality

var compiled = session.getCompiledProgram("degreeCentrality")

var result = compiled.run(G, degreeCentrality)
```

```
{
  "success" : true,
  "canceled" : false,
  "exception" : null,
  "returnValue" : null,
  "executionTimeMs" : 1
}
```

```
%pgql
SELECT v1, v2, v3, e1, e2, e3 FROM connections MATCH (v1) - [e1] -> (v2) - [e2] -> (v1)
WHERE v1.degreeCentrality >= 2 AND v2.degreeCentrality>= 2 AND v3.degreeCentrality>= 2
```

Figure 2 Example of algorithm and query run on a notebook.

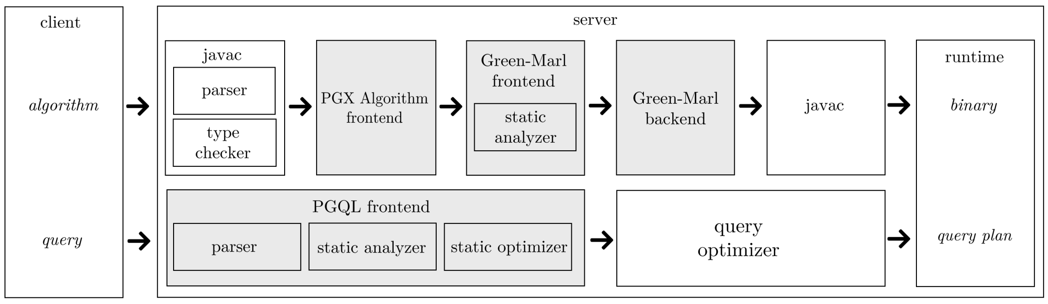


Figure 3 Language processing in the PGX server architecture.

example when running out of memory while processing a large input. The risk of attacks extends beyond language processors to the dependencies they rely on, such as dependencies for reading configurations or for logging[1].

Spoofax' many dependencies increase the risk for vulnerabilities. The *Spoofax* team tracks known vulnerabilities, updates dependencies with fixed versions, and releases new versions of *Spoofax* on a regular basis, as well as on request from its industrial partners. A reduction in third-party dependencies is needed to reduce the risk of vulnerabilities in the *Spoofax* runtime.

▶ **Requirement 4** (Economic Memory Usage). *Language processors typically run continuously to stay available for future requests throughout a server session. Thus, the memory footprint of idle language processors contribute to the overall memory footprint on the server.*

In our experience, *Spoofax* language processors tend to have larger memory footprints than needed, especially when language processors are generated from multiple dependent *Spoofax* projects. These dependencies tend to duplicate the code of the dependee into the dependent project, thus wasting memory. This causes language processors, specifically backends, to be unnecessarily big, as they include all the language definitions of the frontends they depend on, even if the latter are not needed by the backends at runtime. This will be solved in future releases of *Spoofax*, which rely on flexible, composable compiler pipelines.

3 Software (Language) Engineering for Graph Analytics

Domain-specific languages for graph analytics need to be designed, developed, maintained, tested, and released together with the larger graph analytics solution they belong to. Throughout this section, we collect requirements for the engineering of language processors as part of a larger software engineering project and report on our specific experiences with *Spoofax*.

▶ **Requirement 5** (Reusable Language Processors). *Language processors need to be reusable in different settings. For example, a frontend might be combined with different backends for different products. Separated language processor projects allow for a combination of open-source and proprietary projects.*

Spoofax supports multi-project language definitions. We rely on this to split our language definitions into frontend and backend aspects. This allows us to compose language processors into different products. For example, we have separate projects for the *Green-Marl* frontend

[1] `https://nvd.nist.gov/vuln/detail/CVE-2021-44228`

and for each of its backends. We compose the frontend with one of the backends into a compiler we use internally. We recompose the frontend with another backend into another compiler for internal use. We compose the *PGX Algorithm* frontend, the *Green-Marl* frontend, and one of the *Green-Marl* backends into the *PGX Compiler* we include in Oracle products. Separate language projects also allow us to share some projects with our partners, while protecting proprietary solutions in internal projects. For example, the *PGQL* frontend is an open-source project, while the query optimizer is kept in an internal project. Similarly, we share the *Green-Marl* frontend project with academic collaboration partners. We also rely on existing open-source *Spoofax* projects. For example, the *PGX Algorithm* compiler integrates an open-source *Java* syntax definition in *Spoofax*. As discussed earlier, the memory footprint of composed language processors can be improved in future versions of *Spoofax*.

▶ **Requirement 6** (Accessible Language Implementations.). *Language implementations are often maintained by small teams. Team members might only contribute part-time or irregularly to the language development. New team members might join while others leave. This requires language implementations to be as accessible as possible. The source of a bug needs to be quickly found. An additional language construct needs to be easily added.*

Oracle adopted *Spoofax* for its promise of high-level, declarative, multi-purpose language definitions. After a decade of use, we find *Spoofax* delivering on this promise. The size of our language engineering teams fluctuates between a single part-time contributor and up to five full-time contributors. New team members get to know the code base quickly and can typically contribute small fixes after a couple of days and more complex features after a couple of weeks. To train new team members in the use of *Spoofax*, we rely on online material and technical support provided by the *Spoofax* team. Beside the declarative nature of language definitions in *Spoofax*, we find the possibility to modularize language definitions extremely useful to keep our codebase organized and thus easily accessible. We organize the code in hierarchical modules. This helps with maintenance, as the different language constructs are easily located in the project. It also enables the extension of the DSL and its compiler, since we can simply add modules for new language constructs, and have them import existing modules, without affecting the existing language implementation. This has allowed us to efficiently support new backends.

▶ **Requirement 7** (Early Development Feedback). *It is important to get early and quick feedback when implementing a new language feature or improving an existing implementation.*

Spoofax integrates into the *Eclipse* IDE and provides editors for its meta-languages with syntax highlighting, syntax checking, error marking, syntax completion, code formatting, type checking, reference resolving, and hover help. This helps to spot errors early, before building the language projects. Furthermore, the same services are available for the developed languages, allowing us to quickly test our languages in an IDE editor. *Spoofax* also provides menu actions for parsing and applying arbitrary transformations such as formatting, analyzing, and (partial) compilation. In our language projects, we configure several menu actions that perform (partial) compilations, which we trigger manually to quickly test the processing of small examples.

▶ **Requirement 8** (Language Processor Testing). *Adhoc tests can build some initial confidence in the implementation of a language processor, but systematic testing is needed to increase this confidence.*

Spoofax provides the *SPT* (SPoofax Testing) meta-language to specify tests for language definitions. We rely on *SPT* to write unit tests for parsing, static analysis, and compilation. These unit tests typically cover each language construct in separate tests. We organize tests in separate projects, following the project structure and the hierarchical module structure of our compilers. We also use *SPT* to write integration tests in order to ensure complex queries and algorithms are correctly processed. However, these tests only address the language processors as stand-alone tools, and do not consider their integration with larger graph analytics solutions.

To test this integration, we have several tests which compile and run queries and algorithms as part of a toolkit or product. Product integration tests call the compiler for a given runtime and execute a number of algorithms in that runtime, checking for unexpected behavior in the compiler, errors while compiling the generated code into binaries, errors while executing the binary in the runtime, and unexpected results of the executed algorithm. These tests check behavior of built-in algorithms, compilation of custom algorithms, and (in)compatibility of certain language features with specific runtimes.

Customers rely on the performance of built-in and custom algorithms. As such, every change to language implementations or runtimes must be tested for degrading performance. Integration and performance tests are generally hardware-intensive to run, and may require a certain client-server architecture. We run them in a cluster, as part of a regression build triggered when a change is submitted to one of our code repositories. Only when this build succeeds can the change be committed.

▶ **Requirement 9** (Automated Builds). *Continuous development and integration of language processors require automated builds of language processors. These builds need to take dependencies between language projects into account. Tests need to be run as part of regression builds. All automatic builds need to work for stand-alone language processors, but also need to be integrated into overarching builds of toolkits and products.*

Early versions of *Spoofax* mainly focused on the interactive language development experience in IDEs. The building steps for *Spoofax* languages were soon encapsulated in *Maven* builds, which then also could be used to build language projects from command-line tools and to automate builds. However, automated *Spoofax* language builds often felt fragile, and hard to get right for new project setups, particularly for multi-project setups. We currently rely on an intermediate solution to integrate *Spoofax* language builds into our *Gradle* builds. The current work on flexible, incremental compilation pipelines is an important milestone towards improving the automated build situation.

▶ **Requirement 10** (Product Releases). *Language implementations need to be maintained as part of long-term support releases of a product. This includes bug fixes, feature backports, and dependency updates.*

The *Spoofax* team provides multiple stable releases of *Spoofax* per year, but none of these releases provide long-term support. This impacts the long-term support of Oracle products. If language processors in a product are affected by bugs or vulnerabilities introduced by *Spoofax* or its dependencies, we need to update these language processors to work with the latest stable *Spoofax* release. Such an update can be problematic if the *Spoofax* release introduces breaking changes.

4 Conclusion

In this paper, we looked back on how teams at Oracle used *Spoofax* to develop domain-specific languages for large-scale graph analytics for over a decade. We discussed the requirements for language processors in the domain of graph analytics and for their development processes. Our experience and the resulting requirements have motivated some of the development directions of *Spoofax*. *Spoofax* continues to help us to explore language designs, optimization techniques, and compilation patterns in various research projects at Oracle Labs, but also supports us in evolving early research prototypes into solid language processors which become part of Oracle products.

References

1 Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. 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*, ASPLOS XVII, pages 349–362, New York, NY, USA, 2012. Association for Computing Machinery. `doi:10.1145/2150976.2151013`.

2 Lennart C.L. Kats and Eelco Visser. The spoofax language workbench. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '10, pages 237–238, New York, NY, USA, 2010. Association for Computing Machinery. `doi:10.1145/1869542.1869592`.

3 Gabriël Konat. *Language-Parametric Methods for Developing Interactive Programming Systems*. PhD thesis, Delft University of Technology, Delft, Netherlands, November 2019.

4 Gabriël Konat, Lennart Kats, Guido Wachsmuth, and Eelco Visser. Declarative name binding and scope rules. In *Software Language Engineering, 5th International Conference, SLE 2012, Dresden, Germany, September 26-28, 2012, Revised Selected Papers*, volume 7745, pages 311–331, January 2013. `doi:10.1007/978-3-642-36089-3_18`.

5 Neo4j. Cypher query language, 2022. URL: `https://neo4j.com/developer/cypher/`.

6 Oracle. Pgql property graph query language, 2022. URL: `https://pgql-lang.org`.

7 Oracle. Pgx documentation, 2022. URL: `https://docs.oracle.com/cd/E56133_01/latest/reference/analytics/pgx-algorithm.html`.

8 Martin Sevenich, Sungpack Hong, Oskar van Rest, Zhe Wu, Jayanta Banerjee, and Hassan Chafi. Using domain-specific languages for analytic graph databases. *Proc. VLDB Endow.*, 9(13):1257–1268, September 2016. `doi:10.14778/3007263.3007265`.

9 Tinkerpop. Tinkerpop, gremlin, 2022. URL: `https://github.com/tinkerpop/gremlin/wiki`.

10 Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. Pgql: A property graph query language. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, GRADES '16, New York, NY, USA, 2016. Association for Computing Machinery. `doi:10.1145/2960414.2960421`.

11 Eelco Visser. A family of syntax definition formalisms. Technical Report P9706, Programming Research Group, University of Amsterdam, August 1997.

12 Eelco Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, September 1997.

13 Guido H. Wachsmuth, Gabriël D. P. Konat, Vlad A. Vergu, Danny M. Groenewegen, and Eelco Visser. A language independent task engine for incremental name and type analysis. In Martin Erwig, Richard F. Paige, and Eric Van Wyk, editors, *Software Language Engineering*, pages 260–280, Cham, 2013. Springer International Publishing.