EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Mathematisch-Naturwissenschaftliche Fakultät
Wilhelm-Schickard-Institut für Informatik
Database Systems Research Group

Bachelorthesis Computer Science

Extending ByePy with geometric types and operators

Zora Lucia Pidde

11.04.2023

Examiner

Torsten Grust

Co-Examiner

-

Supervisor

Tim Fischer

Zora Lucia Pidde:
Extending ByePy with geometric types and operators
Bachelorthesis Computer Science
Eberhard Karls Universität
From 01.02.2023 to 11.04.2023

Selbständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Bachelorthesis selbständig und nur mit den angegebenen Hilfsmitteln angefertigt habe und dass alle Stellen, die dem Wortlaut oder dem Sinne nach anderen Werken entnommen sind, durch Angaben von Quellen als Entlehnung kenntlich gemacht worden sind. Diese Bachelorthesis wurde in gleicher oder ähnlicher Form in keinem anderen Studiengang als Prüfungsleistung vorgelegt.

Ort, Datum

Zora Lucia Pidde

Acknowledgement

I want to thank my parents, Manuela Pidde and Werner Baumgärtel, for their continuous support throughout my life and many more things. I also thank Steffen Pohl for being there and having an open ear and André Greubel for his friendship.

Abstract

ByePy frontend and *Apfel* backend represent a compiler translating Python code to SQL statements that was developed by the database systems research group from the University of Tübingen. Through this process, the compiler could realise runtime savings in executing code containing both imperative language and SQL. Nonetheless, the *ByePy* currently only supports some basic Python datatypes such as integers, floats, strings, booleans, composite types, and arrays. This thesis enhances the applicability of the frontend by including PostgreSQL's geometric datatypes (point, box, circle, line segment, line, path, and polygon) as well as corresponding operators, functions and typecasts. Preceding the enhancement of the compiler, a Python library is developed to mirror PostgreSQL's geometric behaviour.

Contents

Acknowledgement v

Abstract vii

Acronyms xi

1 Introduction 1

2 Background 3
2.1 Previous Work . . . 3
2.2 Technical Background . . . 4

3 Compilation Rules 7
3.1 Python Library . . . 7
3.2 Typing Rules . . . 9
3.2.1 Constructors & Functions . . . 10
3.2.2 Attribute Access . . . 11
3.2.3 Methods . . . 12
3.2.4 Operators . . . 15
3.3 Lowering Rules . . . 17

4 Practical Demonstation 23

5 Discussion 25

Bibliography 27

Acronyms

ANF Administrative Normal Form
AST Abstract Syntax Tree
GIS Geographic Information Systems
RDBMS Relational Database Management System
SQL Structured Query Language
SSA Static Single Assignment Form
UDF User Defined Function

1 Introduction

Mastery of Structured Query Language (SQL) remains a critical requirement in data management and analysis. In a survey of professional developers, the top 4 named database systems were all based on SQL [1].

SQL belongs to the *declarative* programming paradigm, where the focus is on *describing the problem to be solved* [2]. Then again, many programmers are more familiar with languages of the *imperative* paradigm, where primarily the *way to achieve* a goal is expressed. Moreover, imperative coding concepts like function definition and variable assignment are useful imperative programming constructs that increase code reusability and simplify the programming process. The mismatch in paradigm and relevant coding structures may render working with Relational Database Management Systems (RDBMSs) difficult to many programmers and hence discourage greater use.

In order to facilitate the use of imperative programming concepts in SQL applications the Database Research Group at the University of Tübingen has developed the *Apfel* compiler with a Python frontend *ByePy*. For simplicity, throughout this thesis, the combination of both will be referred to as 'the compiler', while the backend will be explicitly named Apfel-compiler. The compiler translates entire Python programs directly into simple SQL statements. It allows to create data-intensive programs imperatively in Python and obtain programs able to run directly on SQL database systems. This approach results in considerable run time savings compared to approaches alternating between imperative interpretation and SQL execution.

Python is a promising candidate for an imperative input language to the compiler. Surveys show Python to be a popular programming language, ranking in the top 5 in many indices [1, 3, 4, 5]. Data analysis was named as the top purpose for using Python, while Python developers ranked SQL as the top language for web and data science. A widespread system for data administration is PostgreSQL [1, 6]. Python users, in particular, named it as their top database [7]. Therefore, the employed combination of Python and PostgreSQL has great potential. However, *ByePy* currently only supports basic Python language features such as base types, arrays and composite types (detailed description in Chapter 2), while some RDBMSs offer other data types tailored for specific types of problems.

One prominent category of algorithms benefiting strongly from the inclusion of additional types in the compiler is that of computational geometry since the occurrence of large amounts of geometric data requires efficient data management. As RDBMSs are designed to enable this, it is advantageous for geometric algorithms to utilise such a system.

On the other hand, such algorithms often are described imperatively and exhibit complex control flow, which makes direct implementation in SQL difficult. By integrating geometric data

types in *ByePy*, various new applications for the compiler may be enabled through the use of geometric shape constructs. Applications of computational geometry can be found in computer graphics, Geographic Information Systems (GIS), and robotics, amongst others [8]. Computational geometry is hence an indispensable part of modern digital society. Through inclusion of geometric structures in *ByePy*, computations involving large amounts of geometric data could be more easily programmed using imperative Python and could be greatly accelerated, as *ByePy*'s runtime savings might be realised for them.

To enable these advantages, the objective of this thesis is to extend *ByePy* with geometric types. Since the compiler is adapted to PostgreSQL, the geometric types (*point, box, circle, line segment, line, path, polygon*), operators and functions integrated in *ByePy* are those included in PostgreSQL.

In the upcoming chapters, the compiler will be analysed more closely and the steps taken to extend its applicability are shown. First, we give a short overview of the previous work which lead to the compiler's construction and its working mechanisms (Chapter 2). In the next step, we summarise the structure of a newly created Python library matching PostgreSQLs geometric constructs and define type-checking and transformation rules while explaining the design decisions leading to their composition (Chapter 3). A working example illustrates the application of the rules to Python code. The practicality of our *ByePy* extension for relevant algorithms in computational geometry is evaluated in Chapter 4. Finally, the conclusion (Chapter 5) recapitulates the main findings and discusses future options for the compiler with geometric shapes.

2

Background

This chapter gives a brief overview of the previous work that contributed to the development of the compiler. It gives the reader a better understanding of the steps taken to extend the compiler by explaining the motivation for its construction and its operating mechanisms.

2.1 Previous Work

Currently, many solutions to provide imperative features for SQL-based systems are in existence. They range from the definition of User Defined Functions (UDFs) to actually integrating imperative languages with the database [9].
Nonetheless, such solutions are not without problems of their own. Particularly the poor performance of such solutions is lamented [10, 11, 12, 13, 14, 9]. Substantial runtime overhead discourages wider use. The overhead accumulates in practice through many context switches during the execution, where imperative and declarative code needs to be compiled or interpreted in turns. This problem is even more pointed if many database accesses via embedded subqueries are in action. Each subquery execution requires individual instantiation, execution and teardown, even for the same subquery if executed repeatedly as within looping control flow [12]. At the same time, precisely employing a mixture of SQL and imperative code is considered particularly desirable as it improves the readability and maintainability of programs [10].
To preserve the benefits of imperative coding while improving performance, Ramachandra and Colleagues [10] opted to transform entire UDFs into relational algebraic expressions and inline them into the calling query. They modelled a framework named *Froid* which is able to transform complete programs with branching control flow into SQL.
Following their line of work, the Database Research Group of the University of Tübingen developed their own compiler for transforming arbitrary complex and even looping control flow. Throughout several releases, they refined their framework, finally even providing the *ByePy* frontend to compile entire Python programs into plain SQL.
Both frameworks, *Froid* and *ByePy*, achieve runtime savings by avoiding context switches, thus supporting the approach of converting imperative constructs directly to SQL.
Runtime comparisons determining the scale of time savings were executed by all the publications mentioned above. Each of them found substantial runtime improvements to be realised by the use of their respective framework. Particularly noteworthy are [10, 12, 14] as they test their applications on a vast amount of programs. They found speedups from about a factor of two [12] to multiple orders of magnitude [10]. Their results underline the usefulness of their concept, which is to be enhanced to a larger input set in this thesis.

Building upon this work, we aim to enable those benefits for programs involving geometric shapes. To do so, we enable the existing *ByePy*-frontend to translate geometric constructs to SQL. The following section summarises the current state and mechanisms of *ByePy* to emphasise which parts require modification to allow the processing of geometric structures.

2.2 Technical Background

The purpose of this section is to clarify preconditions for the use of the compiler with shapes integration and present its internal composition. The latter promotes an understanding of internal mechanisms, which is a prerequisite for extension.
Technical restrictions must be considered for the use of *ByePy*, and hence our geometric shapes compilation. Most importantly, the PostgreSQL-Version chosen to execute the compiled code must be 11 or higher in order to support geometric types. This prerequisite is not necessary for the use of *ByePy* without shapes integration.

As described in [13, 14, 9], a Python program to be compiled by *ByePy* may use:

- looping and iteration
- control flow statements
- conditional statements
- arbitrarily nested assignment and reference
- lists, indexed access and slicing, stateful list methods
- dictionaries
- specific statements on containers (e.g. delete statement)
- a collection of builtin operators and functions
- falsifiability of builtins
- nested None disambiguation
- embedded read-only queries

Note that the compiler currently does not support the use of recursion, general nested function calls (i.e. "helper functions"), and embedded queries that perform write-operations on the database. Also, the set of assisted builtin functions is restricted to *len, random, abs, ceil, float, floor, copysign, min, max,* and *sqrt*. To enable *ByePy*-transformation, it must be abstained from using other Python builtins.
The current compiler version will be sketched to identify the compiler stages that require additional structure to process the new types and operations. Special attention is hereby given to the description of *ByePy* (see Fig.2.1), which transforms Python input to a mixture of SQL and Static Single Assignment Form (SSA). Control flow is expressed as GOTO statements, while the remainder is emitted as SQL queries. The *Apfel*-compiler then processes this mixture to result in one potentially recursive SQL query. Since the code passed from the frontend to the backend consists, aside from control flow, of SQL queries, the integration of shapes is only necessary for *ByePy*.

For completeness sake, *Apfel* will be shortly outlined too, but for more in-depth explanations the interested reader is advised to refer to the work of Hirn and Grust [12].

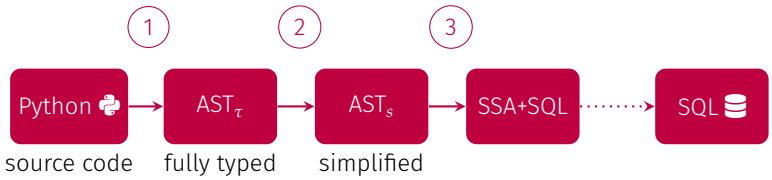

Figure 2.1: ByePy Stages after [9]

ByePy is built around an Abstract Syntax Tree (AST) to manage the data internally. In the last stage of the frontend, AST constructs are mapped to SQL queries.
When Python code first enters *ByePy*, Python constructs are matched on their equivalent *ByePy*-AST types. A mismatch in the internal treatment of types between Python and SQL renders an explicit type-checking step in *ByePy* necessary. Python does not differentiate between basic data types like integer or float and classes [15] and instead comprehends every data as an object. This concerns basic types like integers, collection types like lists and even for functions. Variables are seen as names pointing to the corresponding object of unfixed type. Though since Python Version 3.6 optional typing for variables was introduced, those type annotations are not binding and can be ignored. As the type of a variable is not fixed before runtime, it can be changed by assigning values of different types. Python is thus *dynamically typed* [16]. The target-language SQL, on the other hand, is *statically typed* and requires explicit type definition before runtime [9]. Some Python constructs remain due to Python's untyped design with unclear types, such as function return types [9]. Since types must be known in order to construct valid SQL code, these have to be derived in a type-checking step (see Fig.2.1, step 1). To ensure type safety for SQL, the mentioned type annotations are employed in *ByePy*'s input programs, and explicit type checking is performed within the compiler.
To derive unknown types, the type-checking makes use of known information, e.g. about parameter types. A fully annotated *ByePy*-AST emerges as the first intermediate representation (AST_τ). As the work of this thesis includes several new types, functions, and methods in *ByePy*, type-checking structure needs to be added for those new constructs. A detailed description of these structures is given in Chapter 3.
Subsequent desugaring (see Fig 2.1, step 2) limits the number of constructs to handle in later stages to facilitate further processing. A simplified AST, AST_s, remains. Due to the close and intended likeness of our Python constructs and PostgreSQL's geometric shapes, desugaring is unnecessary for these.
In the next step (lowering, Fig 2.1, step 3), transformation to SQL queries is performed, while control flow is translated to labelled blocks *k* and GOTO statements [9]. For each non-basic type as a method, the arguments are lowered recursively and then included in SQL structure matching the AST construct.
This stage requires supplementary transformation rules to enable *ByePy* to translate geometric

shapes, operators, functions and methods to SQL.

Apfel finalises the translation to complete SQL:1999 code. From SSA, the code is further processed to represent Administrative Normal Form (ANF), and finally trampolined, such that all tail-recursive functions left by the ANF produce one recursive function only. Single tail-recursive functions can be expressed in SQL using WITH RECURSIVE, thus representing valid SQL code. Hence code emitted by the backend consists of SQL queries entirely and can be executed by PostgreSQL.

In the next chapter *ByePy*'s first and third stages are investigated closer since both steps are involved in compiling geometric shapes.

As argued in this chapter, the compiler is a helpful tool for combining imperative algorithms written in Python with RDBMS integration. Therefore, extending *ByePy* with geometric data types is a promising approach to speed up algorithms from the field of computational geometry.

3 Compilation Rules

Rendering PostgreSQLs geometric shapes processable for *ByePy* requires several actions. First, a Python library is needed to allow the use of geometric types and operations in Python. To the best of our knowledge, there currently is no Python library perfectly depicting the shapes, operations, and functions PostgreSQL provides. Hence this library needs to be created in Python. Next, the compiler must be enabled to transform such constructs properly. To achieve this, additional rules must be included in *ByePy*, which instruct the compiler how to handle geometric structures. The relevant stages for these changes are type checking and lowering. In the type-checking section, additional rules were defined to instruct the compiler how to derive the types for our newly introduced Python shapes. The lowering was enhanced by rules describing how a shape construct is to be transformed to yield a matching SQL construct.
This chapter presents the constructed Python library and is thereafter subdivided into rules concerning the derivation of types and rules concerning the translation to PostgreSQL code.

3.1 Python Library

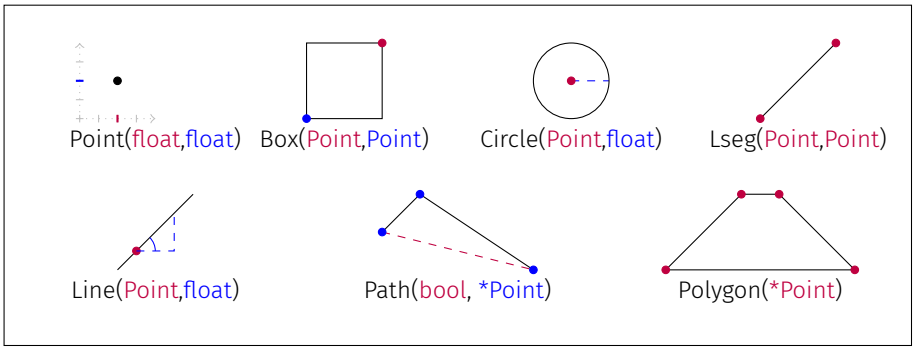

Figure 3.1: Supported Shapes with Constructors in Python

To embed the PostgreSQL geometric support in *ByePy*, the intended shapes, operators, and functions must be replicated for Python first. In order to facilitate the use of the newly constructed library and match PostgreSQL's documentation closely, it was attempted to provide similarity with PostgreSQL and prevalent Python concepts. In many cases, implementation is oriented at PostgreSQL's implementation [17].
Since Python supports objects, and the real-world-constructs shapes can be naturally expressed

as objects, the contained shapes *point, box, circle, line segment, line, path, and polygon* are implemented as object classes.

The target language PostgreSQL provides cast functions and string notation to construct a shape. A *point*, for instance, can be generated using the function point(x,y), inserting two double values, x and y. Alternatively, it can be constructed from other shapes such as *box, circle, line segment* and *polygon*. String notation for a point is point '(x,y)', where x and y are strings that can be cast to double values. In general, the equivalent to PostgreSQL cast functions was chosen as Python constructor, as illustrated in Fig 3.1. Exceptions are *path* and *polygon*, as the available typecasts limit these to only a subset of the shapes that can be expressed by defining their points in string representation. They received a Python constructor, which takes a variable number of points, mimicking the PostgreSQL string notation. Paths also required a boolean status value defining whether the first and last points are connected. Additional PostgreSQL cast functions were implemented as methods or functions.

While considering the object orientation of the project, the representation of PostgreSQL operators as such in Python was pursued. Since Python prohibits defining new operators, functionality that could not be implemented by overloading existing Python operators was realised as method. In cases where these were impracticable, functions were employed. This approach produced minor inconsistencies in functionality. To support orientation, source code documentation is provided in the Python library.

Listing 3.1 presents a snippet of code that will be used for demonstration purposes in the remainder of this chapter. Emphasis is placed on the compiler transformations discussed in the respective sections. The program defines a part of a tiny game where a crow, represented by a *Point*, tries to reach an egg, whereby windows represented by *Line Segments* appear as obstacles along the way (see Fig. 3.2). If the crow collides with windows, it bounces off. The program receives the player's current position, the walk vector and the position of the goal as arguments and calculates the next position under consideration of a possible collision. If the egg is reached, the center of the circle representing it is returned. Keeping our working example as simple as possible, presumptions are that windows are only placed horizontally or vertically, and the crow can not fly far enough in one step for the flight to potentially cross two windows.

```
@to_compile
def play(walk: Point, player: Point, goal: Circle) -> Point:
    nextpos: Point = player + walk
    walk_vec: Lseg = Lseg(player,nextpos)
    collide: Lseg|None = SQL("SELECT wall::lseg FROM obstacles WHERE wall \?# $1;", [walk_vec])
    if collide is not None:
        bounce_pt: Point|None = walk_vec.intersect_point(collide)
        if bounce_pt is not None:
            restwalk: Point = walk - bounce_pt + player
            if collide.horizontal():
                restwalk = Point(restwalk.x, -restwalk.y)
            elif collide.vertical():
                restwalk = Point(-restwalk.x, restwalk.y)
            nextpos = bounce_pt + restwalk
    if goal.contains(nextpos):
        return goal.center
    return nextpos
```

Listing 3.1: Python Game Code

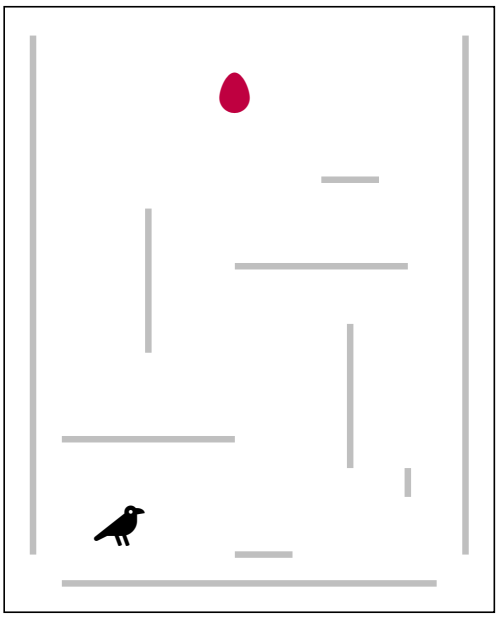

Figure 3.2: Example Game

3.2 Typing Rules

The type-checking stage of *ByePy* consists of rules defining the return type of a statement or expression, based on their identifier and input type(s). The rules concerning our Python shape library by which *ByePy*'s type-checking was extended are presented in this section.
An internal design decision had to be made in relation to the construction of type-checking instructions, that determine validity of input types and derive the expected output type of operations. In PostgreSQL documentation, there is not a single shape that implements a subset of another shape's functionality exclusively. The same holds for the constructed Python library, as it provides exactly the same functionality as PostgreSQL. Hence no shape is a subtype of another shape. Unification types (e.g. point|circle, read as *point or circle*) are not supported by *ByePy* yet. To determine valid input types and corresponding output types two choices remain: formulating instructions for each functionality and each combination of input types or defining artificial supertypes. Such supertypes are specifically tailored for a set of operations that accept exactly the same input types. For code-readability and maintainability it is obviously preferable to use the latter option. An example is the *HasArea*-supertype, which comprises *Box, Circle,* and *Polygon*. The artificial *HasArea*-supertype can, amongst others, be used to determine if both operands to the operator && *(overlap)* are of a type for which this operator is specified (namely Box, Circle, and Polygon).
However, the discussed considerations are of internal consequence only and do not directly affect the user. The formulas presented here do not show such implementation details. For

readability, they use concrete types and case distinction where necessary.
As the type checking does not change Python code aside from assigning matching AST types, the structure of the rules directly reflects the structure of the Python shape library constructed for this thesis. Since AST representation is only needed internally and therefore knowledge of these types is dispensable to the reader, we present the derived types as Python types.
To explain how the type checking instructions are to be read, an example is described in detail.

$:$ is read as *has type*
$<:$ is read as *is subtype of*

Consider the type-checking formula for the constructor *Point*:
If we have an expression e_1 of type τ_1, and an expression e_2 of type τ_2:

$$\Gamma \vdash e_1 : \tau_1$$
$$\Gamma \vdash e_2 : \tau_2$$

And τ_1 and τ_2 are subtypes of type float:

$$\tau_1, \tau_2 <: \text{float}$$

Then the result of function Point(e_1, e_2) is of type *Point*:

$$\Gamma \vdash \text{Point}(e_1, e_2) : \text{Point}$$

Which results in the following formula:

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2 \qquad \tau_1, \tau_2 <: \text{float}}{\Gamma \vdash \text{Point}(e_1, e_2) : \text{Point}} \quad (\textsc{Point})$$

For the sake of readability, the type-checking rules are divided into several segments. Before presenting the actual rules in a segment, a table illuminates the meaning of the PostgreSQL operators the rule applies to.

3.2.1 Constructors & Functions

To present the shape constructors first, we start our summary of typing rules with functions. Constructors are classified as functions in Python. Also, some casts and operators where implemented as functions and will be present in this section.
Following the object-oriented paradigm it was aimed to implement each PostgreSQL operator that could not be represented by a Python operator as a method. However, this could not be realised for each operator as some shapes mutually depended on each other, leading to cross-import conflicts. To avoid this, those operations are implemented as functions.

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2 \qquad \tau_1, \tau_2 <: \text{float}}{\Gamma \vdash \text{Point}(e_1, e_2) : \text{Point}} \text{ (Point)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2 \qquad \tau_1, \tau_2 <: \text{Point}}{\Gamma \vdash \text{Box}(e_1, e_2) : \text{Box}} \text{ (Box)}$$

$$\frac{\Gamma \vdash e : \tau \qquad \tau \in \{\text{Point, Circle}\}}{\Gamma \vdash \text{to\_box}(e) : \text{Box}} \text{ (Box)}$$

$$\frac{\Gamma \vdash e_1 : \text{Point} \qquad \Gamma \vdash e_2 : \tau \qquad \tau <: \text{float}}{\Gamma \vdash \text{Circle}(e_1, e_2) : \text{Circle}} \text{ (Circle)}$$

$$\frac{\Gamma \vdash e : \text{Box}}{\Gamma \vdash \text{to\_circle}(e) : \text{Circle}} \text{ (Circle)}$$

$$\frac{\Gamma \vdash e_1 : \text{Point} \qquad \Gamma \vdash e_2 : \text{Point}}{\Gamma \vdash \text{Lseg}(e_1, e_2) : \text{Lseg}} \text{ (Lseg)}$$

$$\frac{\Gamma \vdash e_1 : \text{Point} \qquad \Gamma \vdash e_2 : \tau \qquad \tau <: \text{float}}{\Gamma \vdash \text{Line}(e_1, e_2) : \text{Line}} \text{ (Line)}$$

$$\frac{\Gamma \vdash e_1 : \text{Point} \qquad \Gamma \vdash e_2 : \text{Point}}{\Gamma \vdash \text{to\_line}(e_1, e_2) : \text{Line}} \text{ (Line)}$$

$$\frac{\Gamma \vdash e_1 : \text{Bool} \qquad \Gamma \vdash e_2 : \text{Point} \qquad \ldots \qquad \Gamma \vdash e_n : \text{Point}}{\Gamma \vdash \text{Path}(e_1, e_2, \ldots, e_n) : \text{Path}} \text{ (Path)}$$

$$\frac{\Gamma \vdash e_1 : \text{Point} \qquad \ldots \qquad \Gamma \vdash e_n : \text{Point}}{\Gamma \vdash \text{Polygon}(e_1, \ldots e_n) : \text{Polygon}} \text{ (Polygon)}$$

$$\frac{\Gamma \vdash e : \tau \qquad \tau \in \{\text{Box, Circle, Path}\}}{\Gamma \vdash \text{to\_polygon}(e) : \text{Polygon}} \text{ (Polygon)}$$

$$\frac{\Gamma \vdash e_1 : \text{int} \qquad \Gamma \vdash e_2 : \text{Circle}}{\Gamma \vdash \text{circle\_to\_poly\_n}(e_1, e_2) : \text{Polygon}} \text{ (Polygon)}$$

OPERATOR	MEANING
a <@ b	a contained in b

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \text{Box} \qquad \tau \in \{\text{Point, Box, Lseg}\}}{\Gamma \vdash \text{contained\_in\_box}(e_1, e_2) : \text{bool}} \text{ (<@)}$$

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \text{Circle} \qquad \tau \in \{\text{Point, Circle}\}}{\Gamma \vdash \text{contained\_in\_circle}(e_1, e_2) : \text{bool}} \text{ (<@)}$$

$$\frac{\Gamma \vdash e_1 : \text{Point} \qquad \Gamma \vdash e_2 : \text{Lseg}}{\Gamma \vdash \text{contained\_in\_lseg}(e_1, e_2) : \text{bool}} \text{ (<@)}$$

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \text{Line} \qquad \tau \in \{\text{Point, Lseg}\}}{\Gamma \vdash \text{contained\_in\_line}(e_1, e_2) : \text{bool}} \text{ (<@)}$$

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \text{Path} \qquad \tau \in \{\text{Point, Path}\}}{\Gamma \vdash \text{contained\_in\_path}(e_1, e_2) : \text{bool}} \text{ (<@)}$$

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \text{Polygon} \qquad \tau \in \{\text{Point, Polygon}\}}{\Gamma \vdash \text{contained\_in\_polygon}(e_1, e_2) : \text{bool}} \text{ (<@)}$$

3.2.2 Attribute Access

PostgreSQL supports retrieving the x- and y-coordinate of a point, the upper right and lower left point defining a box, center and radius from a circle, and deriving the start and endpoints of a

line segment. Therefore, the Python library provides attribute access to the referenced shapes.

$$\frac{\Gamma \vdash e : \text{Point}}{\Gamma \vdash e.\text{x} : \text{float}} \text{ (Point x)} \qquad \frac{\Gamma \vdash e : \text{Point}}{\Gamma \vdash e.\text{y} : \text{float}} \text{ (Point y)}$$

$$\frac{\Gamma \vdash e : \text{Box}}{\Gamma \vdash e.\text{upper\_right} : \text{Point}} \text{ (Box upper right)} \qquad \frac{\Gamma \vdash e : \text{Box}}{\Gamma \vdash e.\text{lower\_left} : \text{Point}} \text{ (Box lower left)}$$

$$\frac{\Gamma \vdash e : \text{Circle}}{\Gamma \vdash e.\text{center} : \text{Point}} \text{ (Circle Center)} \qquad \frac{\Gamma \vdash e : \text{Circle}}{\Gamma \vdash e.\text{radius} : \text{float}} \text{ (Circle Radius)}$$

$$\frac{\Gamma \vdash e : \text{Lseg}}{\Gamma \vdash e.\text{start} : \text{Point}} \text{ (Lseg start)} \qquad \frac{\Gamma \vdash e : \text{Lseg}}{\Gamma \vdash e.\text{end} : \text{Point}} \text{ (Lseg end)}$$

3.2.3 Methods

Since our Python library uses object classes and Python does not support the definition of new operators, the vast majority of all operators from PostgreSQL's geometric shapes are implemented as methods.

OPERATOR	MEANING
@-@ a	length of a
@@ a	center point of shape a
# a	number of points in shape a

$$\frac{\Gamma \vdash e : \tau \qquad \tau \in \{\text{Lseg, Path}\}}{\Gamma \vdash e.\text{len}() : \text{float}} \text{ (@-@)} \qquad \frac{\Gamma \vdash e : \tau \qquad \tau \in \{\text{Box, Circle, Lseg, Polygon}\}}{\Gamma \vdash e.\text{center}() : \text{Point}} \text{ (@@)}$$

$$\frac{\Gamma \vdash e : \tau \qquad \tau \in \{\text{Path, Polygon}\}}{\Gamma \vdash e.\text{n\_points}() : \text{int}} \text{ (\#)}$$

OPERATOR	MEANING
a # b	intersection point of a and b if any
a # b	intersection of two boxes
a ## b	point on b that is closest to a
a <-> b	distance between a and b
a @> b	b contained in a

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau \qquad \tau \in \{\text{Lseg, Line}\}}{\Gamma \vdash e_1.\text{intersect\_point}(e_2) : \text{Optional Point}} \ (\#)$$

$$\frac{\Gamma \vdash e_1 : \text{Box} \qquad \Gamma \vdash e_2 : \text{Box}}{\Gamma \vdash e_1.\text{intersect\_box}(e_2) : \text{Optional Box}} \ (\#)$$

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \text{Point} \qquad \tau \in \{\text{Box, Lseg, Line}\}}{\Gamma \vdash e_1.\text{closest\_point}(e_2) : \text{Point}} \ (\#\#)$$

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \text{Lseg} \qquad \tau \in \{\text{Lseg, Line}\}}{\Gamma \vdash e_1.\text{closest\_point}(e_2) : \text{Optional Point}} \ (\#\#)$$

$$\frac{\Gamma \vdash e_1 : \text{Box} \qquad \Gamma \vdash e_2 : \text{Lseg}}{\Gamma \vdash e_1.\text{closest\_point}(e_2) : \text{ Point}} \ (\#\#)$$

$$\frac{\Gamma \vdash e_1 : \text{Polygon} \qquad \Gamma \vdash e_2 : \text{Circle}}{\Gamma \vdash e_1.\text{dist}(e_2) : \text{float}} \ (\text{<->})$$

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \text{Lseg} \qquad \tau \in \{\text{Box, Line}\}}{\Gamma \vdash e_1.\text{dist}(e_2) : \text{float}} \ (\text{<->})$$

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \text{Point} \qquad \tau <: \text{Geometric}}{\Gamma \vdash e_1.\text{dist}(e_2) : \text{float}} \ (\text{<->})$$

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau \qquad \tau <: \text{Geometric}}{\Gamma \vdash e_1.\text{dist}(e_2) : \text{float}} \ (\text{<->})$$

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \text{Point} \qquad \tau \in \{\text{Box, Circle, Path, Polygon}\}}{\Gamma \vdash e_1.\text{contains}(e_2) : \text{bool}} \ (@\text{>})$$

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau \qquad \tau \in \{\text{Box, Circle, Polygon}\}}{\Gamma \vdash e_1.\text{contains}(e_2) : \text{bool}} \ (@\text{>})$$

OPERATOR	MEANING
a && b	do a and b overlap
a << b	is a (strictly) left to b
a >> b	is a (strictly) right to b
a &< b	does a not extend to the right of b
a &> b	does a not extend to the left of b
a <<| b	is a (strictly) below b
a |>> b	is a (strictly) above b
a &<| b	does a not extend above b
a |&> b	does a not extend below b
a <^ b	is box a below box b (edges allowed to touch)
a >^ b	is box a above box b (edges allowed to touch)

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau \qquad \tau \in \{\text{Box, Circle, Polygon}\}}{\Gamma \vdash e_1.\text{overlap}(e_2) : \text{bool}} \ (\&\&)$$

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau \qquad \tau \in \{\text{Point, Box, Circle, Polygon}\}}{\Gamma \vdash e_1.\text{left\_strict}(e_2) : \text{bool}} \ (\text{<<})$$

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau \qquad \tau \in \{\text{Point, Box, Circle, Polygon}\}}{\Gamma \vdash e_1.\text{right\_strict}(e_2) : \text{bool}} \ (\text{>>})$$

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau \qquad \tau \in \{\text{Box, Circle, Polygon}\}}{\Gamma \vdash e_1.\text{not\_right}(e_2) : \text{bool}} \ (\&\text{<})$$

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau \qquad \tau \in \{\text{Box, Circle, Polygon}\}}{\Gamma \vdash e_1.\text{not\_left}(e_2) : \text{bool}} \quad (\&>)$$

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau \qquad \tau \in \{\text{Point, Box, Circle, Polygon}\}}{\Gamma \vdash e_1.\text{above\_strict}(e_2) : \text{bool}} \quad (|>>)$$

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau \qquad \tau \in \{\text{Box, Circle, Polygon}\}}{\Gamma \vdash e_1.\text{not\_below}(e_2) : \text{bool}} \quad (|\&>)$$

$$\frac{\Gamma \vdash e_1 : \text{Box} \qquad \Gamma \vdash e_2 : \text{Box}}{\Gamma \vdash e_1.\text{above\_touch}(e_2) : \text{bool}} \quad (>\text{^})$$

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau \qquad \tau \in \{\text{Point, Box, Circle, Polygon}\}}{\Gamma \vdash e_1.\text{below\_strict}(e_2) : \text{bool}} \quad (<<|)$$

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau \qquad \tau \in \{\text{Box, Circle, Polygon}\}}{\Gamma \vdash e_1.\text{not\_above}(e_2) : \text{bool}} \quad (\&<|)$$

$$\frac{\Gamma \vdash e_1 : \text{Box} \qquad \Gamma \vdash e_2 : \text{Box}}{\Gamma \vdash e_1.\text{below\_touch}(e_2) : \text{bool}} \quad (<\text{^})$$

OPERATOR	MEANING
a ?# b	do a and b intersect
?- a	is a horizontally aligned
a ?- b	are a and b horizontally aligned
?| a	is a vertically aligned
a ?| b	are a and b vertically aligned
a ?-| b	are a and b perpendicular
a ?|| b	are a and b parallel
a = b	Area equality of a and b (Circle, Box)
a = b	Point number equality of a and b (Path)

$$\frac{\Gamma \vdash e_1 : \text{Line} \qquad \Gamma \vdash e_2 : \text{Lseg}}{\Gamma \vdash e_1.\text{intersect}(e_2) : \text{bool}} \quad (?\#)$$

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau \qquad \tau \in \{\text{Box, Lseg, Line, Path}\}}{\Gamma \vdash e_1.\text{intersect}(e_2) : \text{bool}} \quad (?\#)$$

$$\frac{\Gamma \vdash e_1 : \text{Point} \qquad \Gamma \vdash e_2 : \text{Point}}{\Gamma \vdash e_1.\text{horizontal}(e_2) : \text{bool}} \quad (?\text{-})$$

$$\frac{\Gamma \vdash e_1 : \text{Box} \qquad \Gamma \vdash e_2 : \tau \qquad \tau \in \{\text{Lseg, Line}\}}{\Gamma \vdash e_1.\text{intersect}(e_2) : \text{bool}} \quad (?\#)$$

$$\frac{\Gamma \vdash e : \tau \qquad \tau \in \{\text{Lseg, Line}\}}{\Gamma \vdash e.\text{horizontal}() : \text{bool}} \quad (?\text{-})$$

$$\frac{\Gamma \vdash e : \tau \qquad \tau \in \{\text{Lseg, Line}\}}{\Gamma \vdash e.\text{vertical}() : \text{bool}} \quad (?|)$$

$$\frac{\Gamma \vdash e_1 : \text{Point} \qquad \Gamma \vdash e_2 : \text{Point}}{\Gamma \vdash e_1.\text{vertical}(e_2) : \text{bool}} \quad (?|) \qquad \frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau \qquad \tau \in \{\text{Lseg, Line}\}}{\Gamma \vdash e_1.\text{perpendicular}(e_2) : \text{bool}} \quad (?\text{-}|)$$

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau \qquad \tau \in \{\text{Lseg, Line}\}}{\Gamma \vdash e_1.\text{parallel}(e_2) : \text{bool}} \quad (?||) \qquad \frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau \qquad \tau \in \{\text{Box, Circle}\}}{\Gamma \vdash e_1.\text{area\_eq}(e_2) : \text{bool}} \quad (=)$$

$$\frac{\Gamma \vdash e_1 : \text{Path} \qquad \Gamma \vdash e_2 : \text{Path}}{\Gamma \vdash e_1.\text{n\_points\_eq}(e_2) : \text{int}} \quad (=)$$

$$\frac{\Gamma \vdash e : \tau \qquad \tau \in \{\text{Box, Circle, Path}\}}{\Gamma \vdash e.\text{area}() : \text{float}} \quad (\textsc{Area}) \qquad \frac{\Gamma \vdash e : \text{Box}}{\Gamma \vdash e.\text{diagonal}() : \text{Lseg}} \quad (\textsc{Diagonal})$$

$$\frac{\Gamma \vdash e : \text{Circle}}{\Gamma \vdash e.\text{diameter}() : \text{float}} \quad (\textsc{Diameter}) \qquad \frac{\Gamma \vdash e : \text{Box}}{\Gamma \vdash e.\text{height}() : \text{float}} \quad (\textsc{Height})$$

$$\frac{\Gamma \vdash e : \text{Path}}{\Gamma \vdash e.\text{is\_closed}() : \text{bool}} \quad (\textsc{Path Stat}) \qquad \frac{\Gamma \vdash e : \text{Path}}{\Gamma \vdash e.\text{is\_open}() : \text{bool}} \quad (\textsc{Path Stat})$$

$$\frac{\Gamma \vdash e : \text{Path}}{\Gamma \vdash e.\text{p\_close}() : \text{Path}} \quad (\textsc{Close path}) \qquad \frac{\Gamma \vdash e : \text{Path}}{\Gamma \vdash e.\text{p\_open}() : \text{Path}} \quad (\textsc{Open Path})$$

$$\frac{\Gamma \vdash e_1 : \text{Point} \qquad \Gamma \vdash e_2 : \text{Point}}{\Gamma \vdash e_1.\text{slope}(e_2) : \text{float}} \quad (\textsc{Slope}) \qquad \frac{\Gamma \vdash e : \text{Box}}{\Gamma \vdash e.\text{width}() : \text{float}} \quad (\textsc{Width})$$

$$\frac{\Gamma \vdash e : \text{Box}}{\Gamma \vdash e.\text{lseg}() : \text{Lseg}} \quad (\textsc{Lseg Cast}) \qquad \frac{\Gamma \vdash e : \text{Polygon}}{\Gamma \vdash e.\text{bound\_box}() : \text{Box}} \quad (\textsc{Bounding Box})$$

$$\frac{\Gamma \vdash e_1 : \text{Box} \qquad \Gamma \vdash e_2 : \text{Box}}{\Gamma \vdash e_1.\text{bound\_box}(e_2) : \text{Box}} \quad (\textsc{Bounding Box}) \qquad \frac{\Gamma \vdash e : \text{Lseg}}{\Gamma \vdash e.\text{to\_point}() : \text{Point}} \quad (\textsc{Point Cast})$$

$$\frac{\Gamma \vdash e : \text{Polygon}}{\Gamma \vdash e.\text{to\_circle}() : \text{Circle}} \quad (\textsc{Circle Cast})$$

3.2.4 Operators

Out of the large number of operators defined for geometric shapes in PostgreSQL, only very few could be implemented as actual operators in Python. The reason can be found in Python's prohibition to define new operators. However, basic arithmetic operators like + and *, such as comparisons as ==, >=, < could be defined for the shape classes.
PostgreSQL's meaning for comparators is inconsistent, though. For circle and box, areas are compared. For a path, the number of points is compared, and line segments are compared for the equality of points.
Since using the established equality operator == for comparing path's point number and the area of boxes and circles would be misleading, and equality for boxes and circles was already defined based on attribute equality, these were implemented as methods with meaningful names.

As this was not the case for line segments, they received the befitting operator implementation.

$$\frac{\Gamma \vdash e_1 : \tau \qquad \begin{array}{c} \otimes \in \{+, -, *, /\} \\ \tau \in \{\text{Point, Box, Circle, Path}\} \\ \Gamma \vdash e_2 : \text{Point} \end{array}}{\Gamma \vdash e_1 \otimes e_2 : \tau} \text{ (ARITHMETICS)} \qquad \frac{\Gamma \vdash e_1 : \text{Path} \qquad \Gamma \vdash e_2 : \text{Path}}{\Gamma \vdash e_1 + e_2 : \text{Optional Path}} \text{ (PATH CONCAT)}$$

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau \qquad \tau \in \{\text{Point, Box, Circle, Lseg, Polygon}\}}{\Gamma \vdash e_1 == e_2 : \text{bool}} \text{ (EQUALITY)} \qquad \frac{\begin{array}{c} \otimes \in \{<, >, >=, <=\} \\ \Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau \\ \tau \in \{\text{Box, Circle, Lseg, Path}\} \end{array}}{\Gamma \vdash e_1 \otimes e_2 : \text{bool}} \text{ (COMPARE)}$$

After presenting the relevant formulas for enforcing type checking, listing 3.2 shows a simplified version of the output from this stage for the working example. AST-types, which are only displayed for basic datatypes, are shown in blue. Originally, the compiler also derives more complex types as a conditional type. Inspecting our mini-game, we observe that all expressions have an according AST type assigned, as did the results from applying operators, methods or functions.
Let us examine one example closer:

```
1 bounce_pt = ( walk_vec ::LsegType ). intersect_point ( collide ::LsegType ) ::Optional
      PointType
```

As walk_vec was declared as a line segment before, it clearly is of the type line segment. The argument to the method intersect_point (collide) is also a known line segment, as its status as an optional line segment type was stripped off in a previous condition (if collide is not None) in line 5. The type checking contains instructions for a method named intersect_point, which define valid object types as line segment or line with an argument of the same type. It also specifies this method to return either a Point or None. The variable bounce_point thus is of type Optional Point.

```
1  def play(walk::PointType, player::PointType, goal::CircleType) -> PointType:
2      nextpos = (player ::PointType + walk  ::PointType)  ::PointType
3      walk_vec = Lseg(player::PointType,nextpos::PointType) ::LsegType
4      collide = SQL("SELECT wall::lseg FROM obstacles WHERE wall \?# $1;", [walk_vec::LsegType])
        ::Optional LsegType
5      if (((collide::Optional LsegType) is not None::None)::BooleanType):
6          bounce_pt = (walk_vec::LsegType).intersect_point(collide::LsegType) ::Optional
       PointType
7          if (((bounce_pt::Optional PointType) is not None::None)::BooleanType):
8              restwalk = ((walk ::PointType - bounce_pt ::PointType)::PointType + player ::
       PointType) :: PointType
9              if ((collide ::LsegType).horizontal()::BooleanType):
10                 restwalk = Point(((restwalk::PointType).x)::FloatType,-(((restwalk::PointType).
       y)::FloatType))  ::PointType
11             elif ((collide ::LsegType).vertical()::BooleanType):
12                 restwalk = Point(-(((restwalk::PointType).x)::FloatType),((restwalk::PointType)
       .y)::FloatType)  ::PointType
13             nextpos =   (bounce_pt ::PointType + restwalk ::PointType) :: PointType
14     if ((goal::CircleType).contains(nextpos ::PointType)  ::BooleanType):
15         return ((goal::CircleType).center ::PointType)
16     return (nextpos::PointType)
```

Listing 3.2: Type Checking Step

3.3 Lowering Rules

The actual transformation from Python geometric shapes into PostgreSQL geometric shapes occurs at the lowering stage of *BytePy*. Here the pythonic-oriented code structure is changed to match the desired SQL concepts.
In accordance with [12, 14], we introduce an inference operator $\mapsto_k$ which defines the translation of *BytePy* expressions to SQL, where the label k identifies the block of code in which the current operation is placed. Translations occur in the presence of potential known block definitions s, which may be updated by the operation.
Although for most shapes the transformation is rather straightforward, using geometric casts over existing PostgreSQL types like *circle(point, double precision)*, some shapes require special treatment. *Path* and *polygon* are exceptions since they have no cast that permits to construct the shapes freely from existing types, such as *points* or *line segments* defining the shapes. Both can be cast into each other, given the *path* is closed, and a *polygon* can additionally be cast from a *circle* or a *box*. Another form of constructing a *path* or *polygon* in PostgreSQL is given as defining it in string shape, which allows for a multitude of more freely defined shapes. Therefore the lowering of those shapes can not simply use a constructor and needs more attention. For those, arguments (namely *points*) were lowered to their SQL equivalent, then cast as text and folded together comma-separated using PostgreSQL's concat operator. For the *path*, an additional distinction of cases needs to be made since the closing status of the *path* is expressed solely by using squared brackets or parentheses. To simplify this procedure, *points* are transformed directly into their string-variant.

$$\frac{(e_1^{\text{Py}}, s) \mapsto_k (e_1^{\text{SQL}}, s_1) \qquad (e_2^{\text{Py}}, s_1) \mapsto_k (e_2^{\text{SQL}}, s_2)}{(Point(e_1^{\text{Py}}, e_2^{\text{Py}})) \mapsto_k (point'('\|e_1^{\text{SQL}} :: text\|',' \|e_2^{\text{SQL}} :: text\|')'} \quad \text{(POINT)}$$

$$\frac{(e_1^{\text{Py}}, s) \mapsto_k (e_1^{\text{SQL}}, s_1) \qquad \ldots \qquad (e_n^{\text{Py}}, s) \mapsto_k (e_n^{\text{SQL}}, s_1)}{(Path(True^{\text{Py}}, *e^{\text{Py}})) \mapsto_k (path'('\|e_1^{\text{Py}} :: text\|',' \|\ldots e_n^{\text{Py}} :: text\|')')} \quad \text{(PATH)}$$

$$\frac{(e_1^{\text{Py}}, s) \mapsto_k (e_1^{\text{SQL}}, s_1) \qquad \ldots \qquad (e_n^{\text{Py}}, s) \mapsto_k (e_n^{\text{SQL}}, s_1)}{(Path(False^{\text{Py}}, *e^{\text{Py}})) \mapsto_k (path'['\|e_1^{\text{Py}} :: text\|',' \|\ldots e_n^{\text{Py}} :: text\|']')} \quad \text{(PATH)}$$

$$\frac{(e_1^{\text{Py}}, s) \mapsto_k (e_1^{\text{SQL}}, s_1) \qquad \ldots \qquad (e_n^{\text{Py}}, s) \mapsto_k (e_n^{\text{SQL}}, s_1)}{(Polygon(*e^{\text{Py}})) \mapsto_k (polygon'('\|e_1^{\text{Py}} :: text\|',' \|\ldots e_n^{\text{Py}} :: text\|')')} \quad \text{(POLYGON)}$$

$$\frac{(e_1^{Py}, s) \rightrightarrows_k (e_1^{SQL}, s_1) \quad (e_2^{Py}, s_1) \rightrightarrows_k (e_2^{SQL}, s_2)}{(Box(e_1^{Py}, e_2^{Py})) \rightrightarrows_k (box(e_1^{SQL}, e_2^{SQL})} \quad \text{(Box)}$$

$$\frac{(e_1^{Py}, s) \rightrightarrows_k (e_1^{SQL}, s_1) \quad (e_2^{Py}, s_1) \rightrightarrows_k (e_2^{SQL}, s_2)}{(Circle(e_1^{Py}, e_2^{Py})) \rightrightarrows_k (circle(e_1^{SQL}, e_2^{SQL})} \quad \text{(Circle)}$$

$$\frac{(e_1^{Py}, s) \rightrightarrows_k (e_1^{SQL}, s_1) \quad (e_2^{Py}, s_1) \rightrightarrows_k (e_2^{SQL}, s_2)}{(Lseg(e_1^{Py}, e_2^{Py})) \rightrightarrows_k (lseg(e_1^{SQL}, e_2^{SQL})} \quad \text{(Lseg)}$$

$$\frac{(e_1^{Py}, s) \rightrightarrows_k (e_1^{SQL}, s_1) \quad (e_2^{Py}, s_1) \rightrightarrows_k (e_2^{SQL}, s_2)}{(Line(e_1^{Py}, e_2^{Py})) \rightrightarrows_k (line(e_1^{SQL}, e_2^{SQL})} \quad \text{(Line)}$$

$$\frac{(e^{Py}, s) \rightrightarrows_k (e^{SQL}, s_1)}{(e^{Py}.len()) \rightrightarrows_k (@\text{-}@e^{SQL})} \quad \text{(Length)}$$

$$\frac{(e^{Py}, s) \rightrightarrows_k (e^{SQL}, s_1)}{(e^{Py}.n\_points()) \rightrightarrows_k (\#e^{SQL})} \quad \text{(\# Pt)}$$

$$\frac{(e_1^{Py}, s) \rightrightarrows_k (e_1^{SQL}, s_1) \quad (e_2^{Py}, s_1) \rightrightarrows_k (e_2^{SQL}, s_2)}{(e_1^{Py}.intersect\_point(e_2^{Py})) \rightrightarrows_k (e_1^{SQL} \# e_2^{SQL})} \quad \text{(Intersect Pt)}$$

$$\frac{(e_1^{Py}, s) \rightrightarrows_k (e_1^{SQL}, s_1) \quad (e_2^{Py}, s_1) \rightrightarrows_k (e_2^{SQL}, s_2)}{(e_1^{Py}.intersect\_box(e_2^{Py})) \rightrightarrows_k (e_1^{SQL} \# e_2^{SQL})} \quad \text{(Intersect Box)}$$

$$\frac{(e_1^{Py}, s) \rightrightarrows_k (e_1^{SQL}, s_1) \quad (e_2^{Py}, s_1) \rightrightarrows_k (e_2^{SQL}, s_2)}{(e_1^{Py}.intersect\_point(e_2^{Py})) \rightrightarrows_k (e_1^{SQL} \# e_2^{SQL})} \quad \text{(Intersect Pt)}$$

$$\frac{(e_1^{Py}, s) \rightrightarrows_k (e_1^{SQL}, s_1) \quad (e_2^{Py}, s_1) \rightrightarrows_k (e_2^{SQL}, s_2)}{(e_1^{Py}.intersect\_box(e_2^{Py})) \rightrightarrows_k (e_1^{SQL} \# e_2^{SQL})} \quad \text{(Intersection)}$$

$$\frac{(e_1^{Py}, s) \rightrightarrows_k (e_1^{SQL}, s_1) \quad (e_2^{Py}, s_1) \rightrightarrows_k (e_2^{SQL}, s_2)}{(e_1^{Py}.closest\_point(e_2^{Py})) \rightrightarrows_k (e_1^{SQL} \#\# e_2^{SQL})} \quad \text{(Closest Pt)}$$

$$\frac{(e_1^{Py}, s) \rightrightarrows_k (e_1^{SQL}, s_1) \quad (e_2^{Py}, s_1) \rightrightarrows_k (e_2^{SQL}, s_2)}{(e_1^{Py}.dist(e_2^{Py})) \rightrightarrows_k (e_1^{SQL} <\text{-}> e_2^{SQL})} \quad \text{(Distance)}$$

$$\frac{(e_1^{Py}, s) \rightrightarrows_k (e_1^{SQL}, s_1) \quad (e_2^{Py}, s_1) \rightrightarrows_k (e_2^{SQL}, s_2)}{(e_1^{Py}.contains(e_2^{Py})) \rightrightarrows_k (e_1^{SQL} @> e_2^{SQL})} \quad \text{(Contains)}$$

$$\frac{(e_1^{\text{Py}}, s) \rightrightarrows_k (e_1^{\text{SQL}}, s_1) \qquad (e_2^{\text{Py}}, s_1) \rightrightarrows_k (e_2^{\text{SQL}}, s_2)}{(e_1^{\text{Py}}.overlap(e_2^{\text{Py}})) \rightrightarrows_k (e_1^{\text{SQL}} \&\& e_2^{\text{SQL}})} \quad \text{(Overlap)}$$

$$\frac{(e_1^{\text{Py}}, s) \rightrightarrows_k (e_1^{\text{SQL}}, s_1) \qquad (e_2^{\text{Py}}, s_1) \rightrightarrows_k (e_2^{\text{SQL}}, s_2)}{(e_1^{\text{Py}}.left\_strict(e_2^{\text{Py}})) \rightrightarrows_k (e_1^{\text{SQL}} \ll e_2^{\text{SQL}})} \quad \text{(Left)}$$

$$\frac{(e_1^{\text{Py}}, s) \rightrightarrows_k (e_1^{\text{SQL}}, s_1) \qquad (e_2^{\text{Py}}, s_1) \rightrightarrows_k (e_2^{\text{SQL}}, s_2)}{(e_1^{\text{Py}}.right\_strict(e_2^{\text{Py}})) \rightrightarrows_k (e_1^{\text{SQL}} \gg e_2^{\text{SQL}})} \quad \text{(Right)}$$

$$\frac{(e_1^{\text{Py}}, s) \rightrightarrows_k (e_1^{\text{SQL}}, s_1) \qquad (e_2^{\text{Py}}, s_1) \rightrightarrows_k (e_2^{\text{SQL}}, s_2)}{(e_1^{\text{Py}}.not\_right(e_2^{\text{Py}})) \rightrightarrows_k (e_1^{\text{SQL}} \;\&< e_2^{\text{SQL}})} \quad \text{(Not Right)}$$

$$\frac{(e_1^{\text{Py}}, s) \rightrightarrows_k (e_1^{\text{SQL}}, s_1) \qquad (e_2^{\text{Py}}, s_1) \rightrightarrows_k (e_2^{\text{SQL}}, s_2)}{(e_1^{\text{Py}}.not\_left(e_2^{\text{Py}})) \rightrightarrows_k (e_1^{\text{SQL}} \;\&> e_2^{\text{SQL}})} \quad \text{(Not Left)}$$

$$\frac{(e_1^{\text{Py}}, s) \rightrightarrows_k (e_1^{\text{SQL}}, s_1) \qquad (e_2^{\text{Py}}, s_1) \rightrightarrows_k (e_2^{\text{SQL}}, s_2)}{(e_1^{\text{Py}}.below\_strict(e_2^{\text{Py}})) \rightrightarrows_k (e_1^{\text{SQL}} \ll\!| \;\; e_2^{\text{SQL}})} \quad \text{(Below)}$$

$$\frac{(e_1^{\text{Py}}, s) \rightrightarrows_k (e_1^{\text{SQL}}, s_1) \qquad (e_2^{\text{Py}}, s_1) \rightrightarrows_k (e_2^{\text{SQL}}, s_2)}{(e_1^{\text{Py}}.above\_strict(e_2^{\text{Py}})) \rightrightarrows_k (e_1^{\text{SQL}} \;\; |\!\gg e_2^{\text{SQL}})} \quad \text{(Above)}$$

$$\frac{(e_1^{\text{Py}}, s) \rightrightarrows_k (e_1^{\text{SQL}}, s_1) \qquad (e_2^{\text{Py}}, s_1) \rightrightarrows_k (e_2^{\text{SQL}}, s_2)}{(e_1^{\text{Py}}.not\_above(e_2^{\text{Py}})) \rightrightarrows_k (e_1^{\text{SQL}} \;\&<\!| \; e_2^{\text{SQL}})} \quad \text{(Not Above)}$$

$$\frac{(e_1^{\text{Py}}, s) \rightrightarrows_k (e_1^{\text{SQL}}, s_1) \qquad (e_2^{\text{Py}}, s_1) \rightrightarrows_k (e_2^{\text{SQL}}, s_2)}{(e_1^{\text{Py}}.not\_below(e_2^{\text{Py}})) \rightrightarrows_k (e_1^{\text{SQL}} \;|\&> e_2^{\text{SQL}})} \quad \text{(Not Below)}$$

$$\frac{(e_1^{\text{Py}}, s) \rightrightarrows_k (e_1^{\text{SQL}}, s_1) \qquad (e_2^{\text{Py}}, s_1) \rightrightarrows_k (e_2^{\text{SQL}}, s_2)}{(e_1^{\text{Py}}.below\_touch(e_2^{\text{Py}})) \rightrightarrows_k (e_1^{\text{SQL}} <\!\hat{} \, e_2^{\text{SQL}})} \quad \text{(Box Below)}$$

$$\frac{(e_1^{\text{Py}}, s) \rightrightarrows_k (e_1^{\text{SQL}}, s_1) \qquad (e_2^{\text{Py}}, s_1) \rightrightarrows_k (e_2^{\text{SQL}}, s_2)}{(e_1^{\text{Py}}.above\_touch(e_2^{\text{Py}})) \rightrightarrows_k (e_1^{\text{SQL}} >\!\hat{} \; e_2^{\text{SQL}})} \quad \text{(Box Above)}$$

$$\frac{(e_1^{\text{Py}}, s) \rightrightarrows_k (e_1^{\text{SQL}}, s_1) \qquad (e_2^{\text{Py}}, s_1) \rightrightarrows_k (e_2^{\text{SQL}}, s_2)}{(e_1^{\text{Py}}.intersect(e_2^{\text{Py}})) \rightrightarrows_k (e_1^{\text{SQL}} \ ?\#\ e_2^{\text{SQL}})} \quad \text{(Intersect)}$$

$$\frac{(e^{\text{Py}}, s) \rightrightarrows_k (e^{\text{SQL}}, s_1)}{(e^{\text{Py}}.horizontal()) \rightrightarrows_k (\ ?\text{-}\ e^{\text{SQL}})} \quad \text{(Horizontal)}$$

$$\frac{(e_1^{\text{Py}}, s) \rightrightarrows_k (e_1^{\text{SQL}}, s_1) \qquad (e_2^{\text{Py}}, s_1) \rightrightarrows_k (e_2^{\text{SQL}}, s_2)}{(e_1^{\text{Py}}.horizontal(e_2^{\text{Py}})) \rightrightarrows_k (e_1^{\text{SQL}} \ ?\text{-}\ e_2^{\text{SQL}})} \quad \text{(Horizontal)}$$

$$\frac{(e^{\text{Py}}, s) \rightrightarrows_k (e^{\text{SQL}}, s_1)}{(e^{\text{Py}}.vertical()) \rightrightarrows_k (\ ?|\ e^{\text{SQL}})} \quad \text{(Vertical)}$$

$$\frac{(e_1^{\text{Py}}, s) \rightrightarrows_k (e_1^{\text{SQL}}, s_1) \qquad (e_2^{\text{Py}}, s_1) \rightrightarrows_k (e_2^{\text{SQL}}, s_2)}{(e_1^{\text{Py}}.vertical(e_2^{\text{Py}})) \rightrightarrows_k (e_1^{\text{SQL}} \ ?|\ e_2^{\text{SQL}})} \quad \text{(Vertical)}$$

$$\frac{(e_1^{\text{Py}}, s) \rightrightarrows_k (e_1^{\text{SQL}}, s_1) \qquad (e_2^{\text{Py}}, s_1) \rightrightarrows_k (e_2^{\text{SQL}}, s_2)}{(e_1^{\text{Py}}.perpendicular(e_2^{\text{Py}})) \rightrightarrows_k (e_1^{\text{SQL}} \ ?\text{-}|\ e_2^{\text{SQL}})} \quad \text{(Perpendicular)}$$

$$\frac{(e_1^{\text{Py}}, s) \rightrightarrows_k (e_1^{\text{SQL}}, s_1) \qquad (e_2^{\text{Py}}, s_1) \rightrightarrows_k (e_2^{\text{SQL}}, s_2)}{(e_1^{\text{Py}}.parallel(e_2^{\text{Py}})) \rightrightarrows_k (e_1^{\text{SQL}} \ ?||\ e_2^{\text{SQL}})} \quad \text{(Parallel)}$$

$$\frac{(e_1^{\text{Py}}, s) \rightrightarrows_k (e_1^{\text{SQL}}, s_1) \qquad (e_2^{\text{Py}}, s_1) \rightrightarrows_k (e_2^{\text{SQL}}, s_2)}{(e_1^{\text{Py}}.area\_eq(e_2^{\text{Py}})) \rightrightarrows_k (e_1^{\text{SQL}} = e_2^{\text{SQL}})} \quad \text{(Area Comp)}$$

$$\frac{(e_1^{\text{Py}}, s) \rightrightarrows_k (e_1^{\text{SQL}}, s_1) \qquad (e_2^{\text{Py}}, s_1) \rightrightarrows_k (e_2^{\text{SQL}}, s_2)}{(e_1^{\text{Py}}.n\_points\_eq(e_2^{\text{Py}})) \rightrightarrows_k (e_1^{\text{SQL}} = e_2^{\text{SQL}})} \quad \text{(N Comp)}$$

$$\frac{(e^{\text{Py}}, s) \rightrightarrows_k (e^{\text{SQL}}, s_1)}{(e^{\text{Py}}.area()) \rightrightarrows_k (area(e^{\text{SQL}}))} \quad \text{(Area)}$$

$$\frac{(e^{\text{Py}}, s) \rightrightarrows_k (e^{\text{SQL}}, s_1)}{(e^{\text{Py}}.diagonal()) \rightrightarrows_k (diagonal(e^{\text{SQL}}))} \quad \text{(Diagonal)}$$

$$\frac{(e^{\text{Py}}, s) \rightrightarrows_k (e^{\text{SQL}}, s_1)}{(e^{\text{Py}}.diameter()) \rightrightarrows_k (diameter(e^{\text{SQL}}))} \quad \text{(Diameter)}$$

$$\frac{(e^{\text{Py}}, s) \rightrightarrows_k (e^{\text{SQL}}, s_1)}{(e^{\text{Py}}.height()) \rightrightarrows_k (height(e^{\text{SQL}}))} \quad \text{(Height)}$$

$$\frac{(e^{\text{Py}}, s) \mapsto_k (e^{\text{SQL}}, s_1)}{(e^{\text{Py}}.is\_closed()) \mapsto_k (isclosed(e^{\text{SQL}}))} \text{ (Closed Path)} \quad \frac{(e^{\text{Py}}, s) \mapsto_k (e^{\text{SQL}}, s_1)}{(e^{\text{Py}}.is\_open()) \mapsto_k (isopen(e^{\text{SQL}}))} \text{ (Open Path)}$$

$$\frac{(e^{\text{Py}}, s) \mapsto_k (e^{\text{SQL}}, s_1)}{(e^{\text{Py}}.p\_close()) \mapsto_k (pclose(e^{\text{SQL}}))} \text{ (Close Path)} \quad \frac{(e^{\text{Py}}, s) \mapsto_k (e^{\text{SQL}}, s_1)}{(e^{\text{Py}}.p\_open()) \mapsto_k (popen(e^{\text{SQL}}))} \text{ (Open Path)}$$

PostgreSQL provides array-like access to retrieve data from *points*, *boxes* and *line segments*, which is used to model attribute access. Indexing the shape with '0' returns the first argument and '1' the second argument from the shape. For *circles*, PostgreSQL supplies functions to retrieve center position and radius.

$$\frac{(e^{\text{Py}}, s) \mapsto_k (e^{\text{SQL}}, s_1)}{(e^{\text{Py}}.center) \mapsto_k (center(e^{\text{SQL}}))} \text{ (Circle Access)} \quad \frac{(e^{\text{Py}}, s) \mapsto_k (e^{\text{SQL}}, s_1)}{(e^{\text{Py}}.radius) \mapsto_k (radius(e^{\text{SQL}}))} \text{ (Circle Access)}$$

$$\frac{(e^{\text{Py}}, s) \mapsto_k (e^{\text{SQL}}, s_1)}{(e^{\text{Py}}.x) \mapsto_k (e^{\text{SQL}}[0])} \text{ (Point Access)} \quad \frac{(e^{\text{Py}}, s) \mapsto_k (e^{\text{SQL}}, s_1)}{(e^{\text{Py}}.y) \mapsto_k (e^{\text{SQL}}[1])} \text{ (Point Access)}$$

$$\frac{(e^{\text{Py}}, s) \mapsto_k (e^{\text{SQL}}, s_1)}{(e^{\text{Py}}.upper\_right) \mapsto_k (e^{\text{SQL}}[0])} \text{ (Box Access)} \quad \frac{(e^{\text{Py}}, s) \mapsto_k (e^{\text{SQL}}, s_1)}{(e^{\text{Py}}.lower\_left) \mapsto_k (e^{\text{SQL}}[1])} \text{ (Box Access)}$$

$$\frac{(e^{\text{Py}}, s) \mapsto_k (e^{\text{SQL}}, s_1)}{(e^{\text{Py}}.start) \mapsto_k (e^{\text{SQL}}[0])} \text{ (Lseg Access)} \quad \frac{(e^{\text{Py}}, s) \mapsto_k (e^{\text{SQL}}, s_1)}{(e^{\text{Py}}.end) \mapsto_k (e^{\text{SQL}}[1])} \text{ (Lseg Access)}$$

After the lowering, we can observe that most calculations are replaced by simple SQL queries (compare Listing 3.3). Control Flow is expressed in Terms of GOTOs. As SSA requires each variable to be assigned exactly once, variables have been versioned. In the case of branching control flow, different versions of a variable may be valid depending on the branch taken. In such cases, a ϕ function determines the correct version. Nested branching leads to comparatively many ϕ functions, so the present example has been manually simplified for readability. SQL queries are shown in squared brackets. It can be observed that shapes, functions, operators and methods are replaced by their SQL counterpart.
Back in our sample line, the method intersect_point has been replaced by the binary PostgreSQL-operator # that returns the intersection point of its operands, if any, and else None. The first operand is the object on which the method was defined in Python, and the second operand is the former argument. SSA-serialisation applied a version number to the variable name.

```
1 bounce_pt_1 = [ SELECT walk_vec_1 # collide_1];
```

```
proc play(walk, player, goal):

k1    nextpos_1 = [SELECT player_1 + walk];
      walk_vec_1 = [SELECT lseg(player, nextpos_1)];
      collide_1  = [SELECT "obstacles"."wall" AS "wall"
             FROM obstacles AS "obstacles"
             WHERE "obstacles"."wall" ?# walk_vec_1];
      IF [SELECT collide_1 IS NOT NULL] THEN GOTO k2 ELSE GOTO k8;

k2       bounce_pt_1 = [SELECT walk_vec_1 # collide_1];
         IF [SELECT bounce_pt_1 IS NOT NULL] THEN GOTO k3 ELSE GOTO k8;
k3          restwalk_1 = [SELECT (walk - bounce_pt_1) + player];
            IF [SELECT ?- collide_1] THEN GOTO k4 ELSE GOTO k7;

k4             restwalk_2 = [SELECT (('(' || (((restwalk_1)[0]) :: text || ',')) ((- (
     restwalk_1)[1]) :: text || ')')) :: point];

k5          IF [SELECT ?| collide_1] THEN GOTO k6 ELSE GOTO k7;

k6             restwalk_3 =  [SELECT (('(' || ((- (restwalk_1)[0]) :: text || ',')) (((
     restwalk_1)[1]) :: text || ')')) :: point];

k7          restwalk_4     =  ϕ(restwalk_2, restwalk_3);
            nextpos_2  = [SELECT bounce_pt_1 + restwalk_4];
            GOTO k8;


k8    nextpos_3 =  ϕ(nextpos_1, nextpos_2);
      IF [SELECT goal @> nextpos_3] THEN GOTO k9 ELSE GOTO k10;

k9       RETURN [SELECT center(goal)];

k10   RETURN nextpos_3;
```

Listing 3.3: Lowering Step Simplified Output

4

Practical Demonstation

In order to demonstrate the capabilities of the compiler in practice, three examples from the field of computational geometry were chosen to be implemented using the Python library constructed for this thesis. Those algorithms were transformed entirely into SQL queries using the extended compiler. The algorithms chosen were

- the quickhull algorithm to compute the convex hull over a set of points
- the ear-clipping algorithm for triangulating polygons
- the k-means algorithm for clustering a set of points into k clusters

Figure 4.1 illustrates the input, output and algorithmic processes.

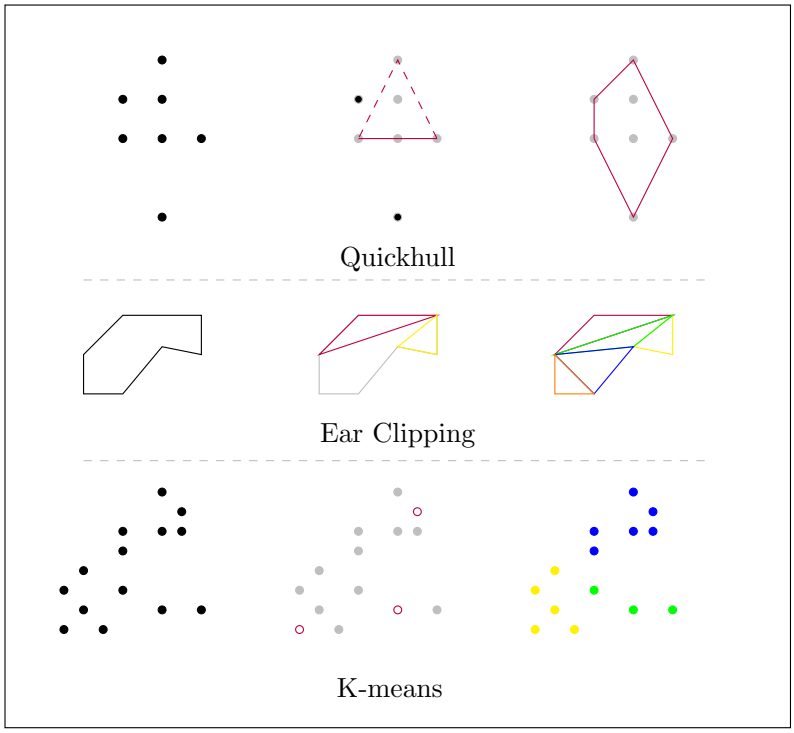

Figure 4.1: Example algorithms: input, intermediate stage, and output

The quickhull algorithm [18, 19, 20] describes a recursive solution for calculating the convex hull over a set of points. The convex hull is the minimum subset of points that covers all

points from the set. In order to use the compiler on it, it had to be rewritten in iterative form since *ByePy* is not yet able to process recursion in Python-Code. The core mechanism of this algorithm is to determine points that are certain to be on the convex hull (as the leftmost or rightmost point) and divide the remainder into sets that are yet covered by the so-constructed intermediate convex hull and such that are not. The remaining sets are further assigned to sets by finding more certain hull points (as the point with the largest distance to one segment of the intermediate hull) until no point can be found outside the intermediate hull anymore. Applications for the convex hull can be found in various domains. Matching and Scheduling are among them, as range searching, statistical trimming [21], and image analysing [22].

Next up is the ear-clipping algorithm. The algorithm iterates a polygon in order to find '*ears*'. An ear, in this context, is defined as a triangle of points on the polygon, which does not contain any other point from the polygon. These are cut from the polygon [23] until only one last triangle remains. The polygon has then been subdivided into a set of triangles. Computer graphics represent a field where such divisions of space are widely used. Modelling 2-dimensional surfaces in 3-dimensional spaces [8] or placement of objects (e.g. of surveillance equipment) in polygonal spaces (e.g. rooms) are example applications.

Last but not least is the k-means algorithm. The k-means algorithm receives a set of points and a number of clusters to produce. Picking k random start points, it then assigns each reference point all points from the set whose distance to that reference point is the minimum distance from all reference points. Afterwards, new reference points are calculated as the mean of the points belonging to a cluster. This is executed iteratively until the cluster assignments do not change anymore. Since unlucky picks for the first reference points may prolong execution time and yield poor results, we oriented the sampling procedure at k++-means [24], where only the first of k reference points is picked at random. In the original work, the next points are sampled with weighted probability. The probability is dependent on the point's distance to the closest sample point. Due to *ByePy*'s restricted input language, we opted to directly pick the next sample point as the one which exhibits the largest minimum distance to the reference points so far.

All three algorithms were successfully implemented using the Python subset applicable to *ByePy*. The compiler translated the programs entirely to SQL and executed them successfully using sample data. It was hence shown that *ByePy* is capable of transforming the newly created Python library for geometric shapes to PostgreSQL. The general applicability of the compiler therefore was extended in a useful way. Please note that this section is concerned with applicability only. Runtime measurements to evaluate the actual speedup are out of the scope of this thesis and remain to be shown.

5 Discussion

The work of this thesis was concerned with enabling the transformation of imperatively written Python programs *containing geometric data* with the compiler consisting of *BytePy* frontend and *Apfel* backend to SQL for PostgreSQL. Although runtime improvement in comparison with other solutions has yet to be shown, an alternative approach for geometric database integration in imperative programs was presented. In the context of this thesis, the practicability for real geometric algorithms was confirmed with polygon triangulation, clustering and calculation of the convex hull. This indicates practicability in actual industrial processes.

USER INFORMATION

Some limitations remain in the current version of *BytePy* that require the user's attention as they restrain the possible input programmes to the compiler.

As Python does not support enforced access protection, the utilisation of internal helper methods and functions by the user can not be restricted completely, even though attempts to compile these with *BytePy* terminate the compiling process with an error. By adding _ or __ in front of an attribute, method or function, an intention for private treatment can be expressed, though no actual prohibition occurs. It is the user's responsibility not to employ constructs marked with __ outside of the library.

Furthermore, PostgreSQL shape parameters can only be retrieved from *points, boxes, circles* and *line segments* for subsequent processing. This is not the case for any other shape. Specifically, the points building a path or polygon can not be extracted anymore for further computations. This poses a severe limitation to the applicability of these shapes.

Moreover, for practical use in implementing geometric algorithms, some very desirable functions are missing in PostgreSQL's geometric constructs. Although quite often necessary when working with *line segments, polygons, paths* and *lines*, there is no function to e.g. determine an angle or decide whether a given *point* is left or right to a *line segment*.

Another serious reservation is the fact that *BytePy* is able to transform a single function only, nested function calls and recursion can not be processed by the compiler despite extensive use. Consequently, code reusability is greatly limited and readability suffers.

WORKAROUND

Some of the listed restrictions can be mitigated by manual workarounds performed by users. If a subset of table entries is to be considered in embedded queries, identifiers from the relation can be kept, e.g. in a list in Python and be passed to queries. If additional computations based on point coordinates for paths or polygons are to be expected, shapes may be stored as points accompanied by identifiers marking their position in the shape and shape membership. For

the use of path or polygon functionality, temporal assembly to the required type is in scope. These solutions require accordingly planned relation design, though.
General nested function calls can be manually inlined by the user, and recursion needs to be bypassed by transforming recursive calls to imperative looping concepts.

FUTURE WORK
Future versions could lighten the burden of manual user workaround by applying some improvements.
In order to provide a broader range of functionality, the Python library can be enhanced by additional methods, accompanied by internal compiler instructions to construct matching functionality in PostgreSQL.
The use of helper functions and recursion can be facilitated by enabling internal automatic inline expansion and supporting transformation from recursive functions to GOTO statements in the compiler.
The compiler that is extended in this thesis generates queries for PostgreSQL from Python code. Additional source- and target platforms might be considered [13] to broaden the applicability further. However, target platforms should provide geometric constructs to benefit from geometric support in the compiler. Examples of RDBMSs providing geometric structure are SQL Server [25] and MySQL [26].

In summary, although *ByePy* currently is applicable to Python geometric programs, its appeal can be improved by implementing additional features such as general nested function calls, recursion and further functionality.

Bibliography

[1] Stackoverflow. *2022 Developer Survey*. URL: https://survey.stackoverflow.co/2022/#technology-most-popular-technologies. (accessed: 01.03.2023).

[2] J. Schmidt H. Ernst and G. Beneken. *Grundkurs Informatik. Grundlagen und Konzepte für die erfolgreiche IT-Praxis - Eine umfassende, praxisorientierte Einführung*. 6th ed. Springer Vieweg, 2016. ISBN: 978-3-658-14633-7.

[3] S. Cass. *Top Programming Languages for 2022*. URL: https://spectrum.ieee.org/top-programming-languages-2022. (accessed: 01.03.2023).

[4] Tiobe Software BV. *Tiobe Index for February 2023*. URL: https://www.tiobe.com/tiobe-index/. (accessed: 01.03.2023).

[5] P. Carbonnelle. *PYPL PopularitY of Programming Language*. URL: https://pypl.github.io/PYPL.html. (accessed: 01.03.2023).

[6] P. Carbonnelle. *TOPDB Top Database Index*. URL: https://pypl.github.io/DB.html. (accessed: 01.03.2023).

[7] Jetbrains s.r.o. *Python Developers Survey 2021 Results*. URL: https://lp.jetbrains.com/python-developers-survey-2021/. (accessed: 01.03.2023).

[8] M. Overmars M. de Berg M. van Kreveld and O. Schwarzkopf. *Computational Geometry. Algorithms and Applications*. 2nd ed. Springer, 2000. ISBN: 3-540-65620-0.

[9] T. Fischer. "To Iterate Is Human, to Recurse Is Divine — Mapping Iterative Python to Recursive SQL". In: *BTW 2023*. Ed. by Birgitta König-Ries et al. Gesellschaft für Informatik e.V., 2023. DOI: 10.18420/BTW2023-73.

[10] K. Ramachandra et al. "Optimization of Imperative Programs in a Relational Database". In: *Proc. VLDB Endow.* 11.4 (2017), pp. 432–444.

[11] D. Hirn C. Duta and T. Grust. "Compiling PLSQL Away". In: 0 (2019), pp. 0-0.

[12] D. Hirn and T. Grust. "One WITH RECURSIVE is Worth Many GOTOs". In: *Proceedings of the 40th ACM SIGMOD Int'l Conference on Management of Data (SIGMOD 2021), Xi'an, Shaanxi, China, June 2021*. 2021.

[13] D. Hirn T. Fischer and T. Grust. "Snakes on a Plan". In: *Proceedings of the 41st ACM SIGMOD Int'l Conference on Management of Data (SIGMOD 2022), Philadelphia, PA, USA, June 2022*. 2022.

[14] T. Fischer. "ByePy: Compilation of Python to SQL". MA thesis. University of Tuebingen, 2022.

[15] M. Kofler. *Python. Der Grundkurs*. 1st ed. Rheinwerk Computing, 2019. ISBN: 978-3-8362-6679-6.

[16] J. Lehtosalo G.van Rossum and Ł. Langa. *Pep 484 - Type Hints*. URL: https://peps.python.org/pep-0484/#non-goals. (accessed: 10.04.2023).

[17] PostgreSQL Global Development Group. *PostgreSQL Source Code. geo_ops.c.* URL: https://doxygen.postgresql.org/geo__ops_8c_source.html#l02775. (accessed: 02.03.2023).

[18] W. Eddy. "A new convex hull algorithm for planar sets". In: *ACM Transactions on Mathematical Software* 3 (1977), pp. 398–403.

[19] A. Bykat. "Convex hull of a finite set of points in two dimensions". In: *Information Processing Letters* 7 (1978), pp. 296–298.

[20] P.J. Green and B.W. Silverman. "Constructing the convex hull of a set of points in the plane". In: *Computer Journal* 22 (1979), pp. 262–266.

[21] J. Hershberger and S. Suri. "Applications of a semi-dynamic convex hull algorithm". In: *BIT* 32 (1992), pp. 249–267.

[22] W. E. Snyder and D. A. Tang. "Finding the Extrema of a Region". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-2.3 (1980), pp. 266–269. DOI: 10.1109/TPAMI.1980.4767016.

[23] H. Everett X. Kong and G. Toussaint. "The Graham Scan Triangulates Simple Polygons". In: *Pattern Recognition Letters* 11 (1990), pp. 713–716.

[24] D. Arthur and S. Vassilvitskii. "K-Means++: The Advantages of Careful Seeding". In: *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '07. New Orleans, Louisiana: Society for Industrial and Applied Mathematics, 2007, pp. 1027–1035. ISBN: 9780898716245.

[25] Microsoft. *Spatial Types - geometry (Transact-SQL)*. URL: https://learn.microsoft.com/en-us/sql/t-sql/spatial-geometry/spatial-types-geometry-transact-sql?view=sql-server-ver16. (accessed: 06.03.2023).

[26] Oracle Corporation. *MySQL Reference 8.0 Manual. 12.17.1 Spatial Function Reference*. URL: https://dev.mysql.com/doc/refman/8.0/en/spatial-function-reference.html. (accessed: 06.03.2023).