EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

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

Masterthesis Computer Science

ByePy: Compilation of Python to SQL

Tim Fischer

30.09.2022

Examiner

Prof. Dr. Torsten Grust

Co-Examiner

Prof. Dr. Klaus Ostermann

Supervisor

Denis Hirn

Tim Fischer:
ByePy: Compilation of Python to SQL
Masterthesis Computer Science
Eberhard Karls Universität
From 01.04.2022 to 30.09.2022

Selbständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Masterthesis 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 Masterthesis wurde in gleicher oder ähnlicher Form in keinem anderen Studiengang als Prüfungsleistung vorgelegt.

Ort, Datum

Tim Fischer

Disclaimer

This thesis is an extended version of a demo paper — *Snakes on a Plan: Compiling Python Functions into Plain SQL Queries* [1] — published in the proceedings of the SIGMOG '22 conference. I was the paper's main author and was helped along the way by my co-authors Denis Hirn and Torsten Grust. In the spirit of transparency and academic integrity, I felt it necessary to declare this properly from the start.

Abstract

Databases have become ubiquitous in the world of application development and have found their way into the toolbelt of many python developers [2]. Sadly, many of these developers still struggle with the database world's de facto lingua franca SQL. This is no fault of their own; SQL differs drastically from *imperative* Python programming in its *declarative* nature. The struggle with SQL has led to many writing code interweaving complex control flow and database access. Such code exhibits poor performance compared to even the most complicated queries. The constant context switches between the Python interpreter and the underlying database add up quickly, resulting in a significant amount of time being wasted on overhead that does not further the computation itself. To address this issue, we present ByePy — a Python to SQL compiler, which can compile entire Python functions with arbitrary control flow into plain recursive SQL:1999 queries. Compilation with ByePy is rewarded by runtime speedups of up to an order of magnitude.

Contents

Acronyms

ANF *Administrative Normal Form*
CC *Cyclomatic Complexity*
CFG *Control Flow Graph*
CTE *Common Table Expression*
DBMS *Databases Management System*
ISO *International Organization for Standardization*
JSON *JavaScript Object Notation*
ORM *Object Relational Mapper*
SF *Scaling Factor*
SFW *Select-From-Where*
SQL *Structured Query Language*
SSA *Static Single Assignment*
UDF *User-Defined Function*

1

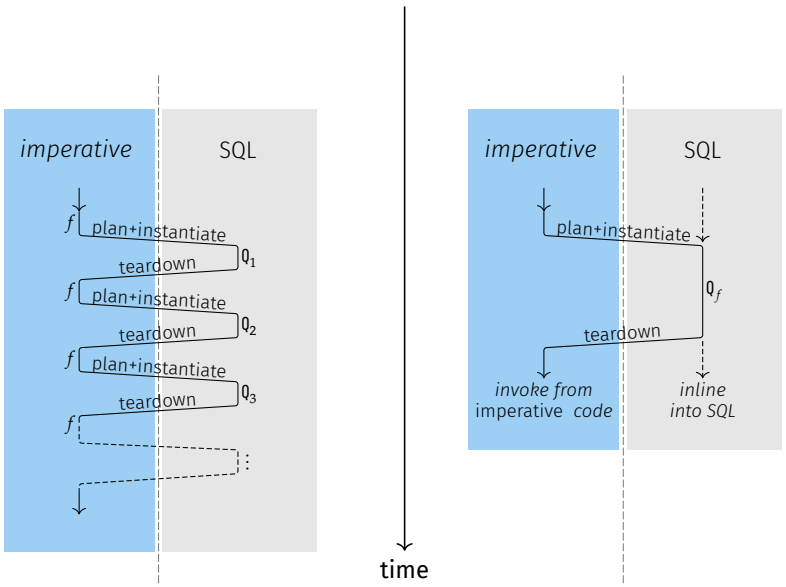
Introduction

Databases have become indispensable. No matter what application you look at — be it in games, web, fintech, *etc.* — chances are it uses a database. Though their applications are diverse, they all build on just a handful of core technologies. The best example is the ubiquity of the relational data model and its defacto lingua franca: the *Structured Query Language* (SQL). But this ubiquity has led to SQL becoming more and more complex to cover evermore use cases — even going so far as to becoming turing complete with the introduction of the SQL:1999 standard [3]. This ever-increasing complexity of SQL has been a gatekeeper to database domain knowledge, making said knowledge a rarity amongst developers, data analysts, and data scientists. And though rare, this knowledge has become an invaluable skill thanks to today's digital world's relentless growth and globalization, resulting in ever larger datasets and reliance upon them for important decisions.

One reason for the apparent complexity of SQL and the resulting lack of domain knowledge in the broader developer community is a difference in language paradigms between SQL and other programming languages. SQL is a *declarative* language in which one describes *what* the result of a computation should be. In stark contrast, most popular languages today are *imperative* languages that focus on describing *how* a result is computed. These paradigms require entirely different thinking when developing and debating solutions. And the *declarative* one often comes short in today's computer science and engineering education.

To make SQL more "*appealing*" to the average developer, many layers of abstractions have been built on top of it to this day — *e.g.*, *Object Relational Mapper*s (ORMs) [4], and query builders. These abstractions primarily focus on embedding the functionality of SQL within the syntax and semantics of a given host language. This kind of abstraction allows users to skip studying the *declarative* nature of SQL and supplement its lack of *imperative* control flow features with the ones provided by the host language. But such an approach leads to inherently slow database access patterns — the constant context switching to and from the database during a computation introduces additional overhead, see Figure 1.1a for a visual depiction.

"*Move your computation close to the data*" [5] is decades-old advice that describes an orthogonal approach. Rather than embedding database access into control flow, embed the control flow into database access — *i.e.*, perform as much computation inside the database as possible, thus minimizing the number of database accesses required in total — see Figure 1.1b for a visual depiction. The main issue with this approach is that it often requires expert knowledge of SQL and *declarative* problem-solving. As such, its benefits often elude many application developers who instead opt to retain the complex control flow and computations in a language they feel comfortable maintaining.



(a) *Imperative* code for function f invokes SQL queries Q_i.

(b) With function f compiled to pure SQL.

Figure 1.1: Interplay of an *imperative* language and an SQL engine. To the right of boundary ⋮, we are inside the *Databases Management System* (DBMS) process.

Embedding complex control flow into database accesses also helps minimize the amount of data needed to be transferred from and to a database. For example, when developers want to use fancy algorithms, they will likely only find their *imperative* form — think, textbook pseudo-code. Translating these into *declarative* SQL is far from trivial in most cases. As such, developers often default to just translating the pseudo-code into their host language and then wrapping or interweaving that implementation with database accesses to retrieve the required data. Depending on the selected algorithm and the environment in which it will be deployed, such implementations move significant amounts of data over a network.

Loading or materializing all data onto a local machine — be it in memory or on disk — is a typical pattern in data science code. Such code is often the direct result of the developers being uncomfortable with SQL and opting to use other data representations — *e.g.*, n-dimensional arrays [6] or data frames [7, 8]. Because of this, much work has gone into making these representations more efficient — for example, dask [9] allows operations on data frames in Python [7] to make use of the parallel computing ability of modern computer architectures. Some of these tools even implement ideas and concepts from the database world — for example, dask has a plan optimizer which allows chained operations to be performed more efficiently. And though such tools often implement more space-efficient storage models — *e.g.*, Apache Parquet [10] — this does not alleviate the issue of materializing significant amounts of data in memory or on disk.

Extracting data from databases can pose another problem, that being the issue of data secu-

rity and data privacy laws. For example, if an American company wants to run some computations on personal data of their European users — which, at the date of writing this thesis, needs to be stored in Europe as the data privacy guarantees in the US do not suffice [11] — they can run into legal ramifications, if they opt to move the data to their US-based machines. A typical fix for this dilemma is running the computations on Europe-based machines, which requires additional infrastructure on an entirely different continent. Tough such infrastructure has admittedly become a lot more accessible thanks to cloud providers like Google and Amazon, it still remains an additional cost that some may not be willing to pay.

All in all, there is an enormous incentive to perform data-heavy computations wherever the data is stored. To make running computations in databases more appealing, most *Databases Management Systems* (DBMSs) today implement some form of user-definable computation — *e.g.*, *User-Defined Functions* (UDFs) — and also some form of dedicated procedural — read *imperative* — languages to make writing these computations more natural for the average developer — *e.g.*, PL/SQL[12], T-SQL[13], PL/pgSQL[14]. Some DBMSs even go a step further and integrate runtimes of existing languages into the database engine, enabling developers to choose the language they are most comfortable with — *e.g.*, Python [15, 16, 17], Java [18], JavaScript [19].

Common developer wisdom argues that using database language integrations is the root cause of many runtime troubles. Some even go so far as to say that using UDFs for computations exhibiting complex control flow with many intermediate queries is generally a bad idea [20, 21, 22, 23, 24, 25]. This is driven by the horrendous performance of UDFs in most DBMSs. Much work has gone into reconciling this dissonance between the promises of UDFs and their lacking performance. To date, there are quite a few optimization approaches — improve the performance of the runtimes embedded in the DBMSs or improve the general performance SQL-based UDFs [23, 25].

The most interesting UDF-optimization approach for this thesis is the line of work kickstarted by Karthik Ramachandra and his colleagues' work on *froid* [20] — which introduced a method of inlining procedural T-SQL code that makes use of simple branching into SQL queries. The Database Research Group at the University of Tübingen later extended Karthik's idea to cover turing complete code via generic loop-constructs — *e.g.* `for`- and `while`-loops [21, 22, 24] — through the use of recursive *Common Table Expressions* (CTEs). The group developed a chain of compilations steps[1], for the first of which this thesis introduces an alternative to support Python.

By the latest accounts, Python is amongst the five most popular programming languages in use today [26, 27]. Python's three primary fields of use are *web development*, *data science* and *machine learning*; all of these fields make use of large datasets in some way or another. In the world of highly efficient off-the-shelf solutions for managing such datasets, one remains king — the database. Communicating with and using a database, however, can be far from trivial, leading many developers, data analysts, and data scientists to default to data management and processing libraries and tools with better ergonomics — *e.g.*, NumPy [6], pandas [7], *etc.*

Python's popularity and its widespread use in data-intensive fields like data science and finance make it the perfect language to find optimizations for. Any speedup of code over database

[1] The original compilation steps — except for the first one — will be explained in Section 2.4.

resident data written in Python will find broad applicability as most Python developers use databases, and barely any of the other existing big data solutions [2].

To enhance the experience and performance of programming with databases in Python we will adapt the compiler toolchain in [24] with a new Python frontend. To that end, we will

1. define a minimal useful Python subset — the ByePy grammar — that allows for computation over database resident data that requires minimal SQL knowledge to use,
2. develop a type checking framework for the ByePy grammar to ensure type correctness during compilation,
3. introduce a set of inference rules that can compile code using the ByePy grammar into a *Control Flow Graph* (CFG) representation — the first intermediate representation of the existing compiler tool chain, and last but not least
4. we will also evaluate this approach and also compare it the likes of PL/Python and PL/SQL.

2

Interna

Translating one language to another is by no means trivial. Especially when the core semantics of the source and target languages differ by such an extreme degree, as is the case in this thesis. Though Python is a multi-paradigm language, its core semantics are primarily *imperative* — *i.e.*, each program, function, method, you name it, is a clearly defined sequence of instructions that describe *how something is computed*. In stark contrast to that SQL is a *declarative* language in which the primary focus lies on describing *what is computed*.

We are in luck, as the translation between these two opposing paradigms — *imperative* vs. *declarative* — has been extensively studied throughout the short history of computer science. In the even shorter history of SQL as a compilation target, one method stands out as a possible candidate to base BYEPY on. That method is the reverse application of the steps that you would commonly find in a machine code compiler for a functional language — *i.e.*, by transforming a Python program into (1) a CFG, (2) performing Φ-placement to achieve *Static Single Assignment* (SSA) form, (3) turning `GOTO`s in function calls to get to *Administrative Normal Form* (ANF), (4) trampolining all functions, and finally (5) translating that to SQL.

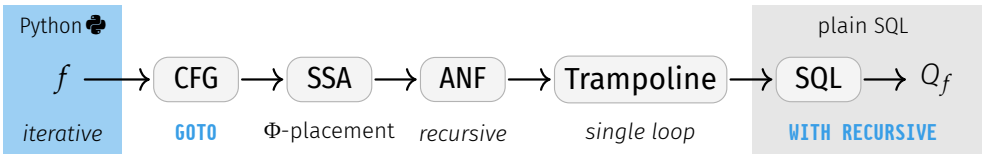


Figure 2.1: Compiler stages and intermediate program forms.

In this thesis, we will focus on the first step in this chain of transformations — *i.e.*, the translation of Python programs into CFGs. Before we get to the translating part in Section 2.3, we will establish the grammars we will be translating between in Section 2.1 and Section 2.2. And to close out, we will also briefly discuss the latter four steps in Figure 2.1 as they were described by Denis Hirn and colleagues [21, 22, 24] — though this is just for completeness' sake and some metaphorical "*meat*" will be missing.

2.1 Sensible Python Subset

Compiling all of Python's quirks — think, coroutine magic or the extensive data model — is complicated and very much outside this thesis's scope. So let us start by exploring and defining a sensible subset for computations over database resident data while maintaining at least some

semblance of *pythonic* code. To get the ball rolling, look at the example in Listing 2.1 — it contains most of the features ByePy supports and gives an excellent overview of how ByePy-compatible code looks and feels.

```
@to_compile
def order_kept_waiting(suppkey: int, orderkey: int) -> bool:
    blame: bool = False  # is suppkey to blame?
    multi: bool = False  # does this order have multiple suppliers?

    lis: list[ShortLineItem] = SQL(
        """
        SELECT array_agg(( l.l_orderkey
                         , l.l_suppkey
                         , l.l_commitdate
                         , l.l_receiptdate
                         ) :: ShortLineItem)
          FROM lineitem AS l
          WHERE l.l_orderkey = $1
        """,
        [ orderkey ]
    )

    for li in lis:
        multi = multi or li.sl_suppkey != suppkey
        if li.sl_receiptdate > li.sl_commitdate:
            if li.sl_suppkey != suppkey:
                return False
            else:
                blame = True

    return multi and blame
```

Listing 2.1: Check if a TPC-H order ist kept waiting.

As you can see, we aim to support control flow constructs you will commonly find throughout Python code — including, but not limited to, iteration via loop statements, early exiting of loops, branching statements, and stateful variable reassignment. Further, we are also shooting for a pythonic expression syntax by including attribute access via dot-notation and a simple function call to facilitate query execution. And last but not least, we also require type annotations where ever the type of an expression is not inferrable — we will get to why exactly this is needed and how this information is used in the compiler in Section 2.1.2.

2.1.1 ByePy Grammar

When formally describing translations between languages, it is indispensable to clarify their respective grammars. So let us start with the Python grammar — rather, the ByePy grammar. The grammar is strictly a subset of the complete Python grammar, as finding translation rules to mirror all of Pythons quirks and semantics is far beyond the scope of this thesis.

```
F := @to_compile ↲ def v(v: τ, … , v: τ) -> τ: ↲ s                function definition
v := ⟨identifier⟩                                                 variable/function name
τ := int | float | str | bool                                     simple types
   | list[τ]                                                      list/array types
   | Optional[τ] | (τ | None)                                     nullable types
   | ⟨composite type⟩                                             composite types
```

(a) Function, Identifiers, Types

```
s := e                                                            bare expression
   | v = e                                                        assignment
   | v: τ = e                                                     annotated assignment
   | v ⊖ e                                                        augmented assignment
   | if e: ↲ s [ ↲ else: ↲ s]                                     conditional statement
   | if v is None: ↲ s [ ↲ else: ↲ s]                             Optional disambiguation
   | for v in range(e): ↲ s                                       onesided range loop
   | for v in range(e, e): ↲ s                                    twosided range loop
   | for v in e: ↲ s                                              array loop
   | while e: ↲ s                                                 while loop
   | break                                                        loop break
   | continue                                                     loop continue
   | return e                                                     return
   | return                                                       bare return
   | s ↲ s                                                        statement sequence
   | f_s                                                          stateful methods
f_s := v.extend(e)                                                stateful list concat
   | v.append(e)                                                  stateful list append
```

(b) Statements

```
e := v                                                            variable literal
   | ⊘e                                                           unary operator
   | e ⊗ e                                                        binary operator
   | SQL("Q")                                                     non-parameterised query
   | SQL("Q", [e, … , e])                                         parameterised query
   | e[e]                                                         array subscript
   | e[e:] | e[:e] | e[e:e]                                       array slicing
   | e.v                                                          attribute access
   | e.m(e, … , e)                                                method call
   | f(e, … , e)                                                  function call
   | e if e else e                                                conditional expression
   | e if v is None else e                                        "inline" Optional disambiguation
   | l                                                            literal (floats, ints, Nones, …)
f := len | random | abs | ceil | float | floor
   | copysign | min | max | sqrt
   | ⟨composite type⟩                                             composite type constructor
m := pop                                                          stateful list pop
```

(c) Expressions

Figure 2.2: Python language subset covered by the compiler.

The set of supported functions is not a strict subset of Python's built-in functions. It also includes some from the `math` library and the constructors for composite types — i.e., object constructors. But of all those, we have to put the spotlight v`.pop()`, v`.append(`e`)`, and v`.extend(`e`)`. These contain side effects! They transform a given list without notice of this via their return value. But not only that, they allow for writing idiomatic Python — *i.e.*, pythonic — code to implement common stack- and queue-based algorithms.

2.1.2 Typing rules

As alluded to earlier, the ByePy grammar uses explicit type annotations. We use these to make the work of the compiler easier — and arguably the programmer's life. Though, in general, types are something Python itself doesn't care about until runtime — said differently, you can write legal Python, in extension ByePy, code and that produces type errors at runtime.

```
1 + "two"            # TypeError: unsupported operand type(s) for +: ...
[].extend(1)         # TypeError: 'int' object is not iterable
floor("a number?!")  # TypeError: must be real number, not str
```

Listing 2.2: Grammatically legal ByePy that produces type errors

This issue is not just something that plagues Python; we expect at least some semblance of type safety from the programming languages we use. Many languages solve this by performing a type checking step during the compilation in which some statically declared types are used to infer type correctness over the remainder of the program.

In contrast, Python implements *duck typing* — *"if it looks like a duck and quacks like a duck, it must be a duck"* [28]. To put it bluntly, there are no type guarantees in Python; all type errors happen at runtime. Recently there has been a surge of development tooling to address this — type annotations [29] and static type checkers [30, 31] using them have become a staple in the Python world [2, Tools and Features for Python Development].

We can take a page from the Python type checking playbook and use type annotations to help ensure type correctness and use that to make our compilation rules behave better. Or at least to avoid defaulting to *"garbage in, garbage out"* and be good Python citizens by following the Zen of Python [32] — "errors should never pass silently." To do so, we need a well-defined set of typing rules — see Figure 2.3.

$$\frac{\Gamma, v_1 : \tau_1, \ldots, v_n : \tau_n \vdash s : \tau_r}{\Gamma \vdash \texttt{@to\_compile} \hookleftarrow \texttt{def}\ v(v_1 : \tau_1, \ldots, v_n : \tau_n)\ \texttt{->}\ \tau_r\texttt{:} \hookleftarrow s : \tau_r}\ \text{(Function Definition)}$$

(a) Function Definition

$$\frac{}{\Gamma \vdash \langle \textit{numeric literal} \rangle : \texttt{float}}\ \text{(Float)} \qquad \frac{}{\Gamma \vdash \langle \textit{numeric literal} \rangle : \texttt{int}}\ \text{(Integer)} \qquad \frac{}{\Gamma \vdash \langle \textit{boolean literal} \rangle : \texttt{bool}}\ \text{(Boolean)}$$

$$\frac{\Gamma \vdash e : \tau \qquad \Gamma \vdash [\textit{rest}] : \texttt{list}[\tau]}{\Gamma \vdash [e, \textit{rest}] : \textit{list}[\tau]}\ \text{(List)} \qquad \frac{}{\Gamma \vdash [] : \texttt{list}[\bot]}\ \text{(Empty List)} \qquad \frac{}{\Gamma \vdash \langle \textit{string literal} \rangle : \texttt{str}}\ \text{(String)}$$

$$\frac{}{\Gamma \vdash \varepsilon : \top}\ \text{(Empty Expression)} \qquad \frac{}{\Gamma \vdash \texttt{None} : \texttt{Optional}[\bot]}\ \text{(None)} \qquad \frac{}{\Gamma, v : \tau \vdash v : \tau}\ \text{(Variable)}$$

(b) Literals

$$\frac{\texttt{Q}^{\texttt{SQL}}[] : \tau}{\Gamma \vdash \texttt{SQL}(\texttt{"Q}^{\texttt{SQL}}\texttt{"}) : \tau}\ \text{(Non-parameterised Query)} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \ \ldots\ \Gamma \vdash e_n : \tau_n \qquad \texttt{Q}^{\texttt{SQL}}[e_1 : \tau_1, \ldots, e_n : \tau_n] : \tau}{\Gamma \vdash \texttt{SQL}(\texttt{"Q}^{\texttt{SQL}}\texttt{"}, [e_1, \ldots, e_n]) : \tau}\ \text{(Parameterised Query)}$$

(c) Queries

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{return}\ e : \tau}\ \text{(Return)} \qquad \frac{}{\Gamma \vdash \texttt{return} : \texttt{None}}\ \text{(Bare Return)} \qquad \frac{\Gamma \vdash e : \tau \qquad \Gamma, v : \tau \vdash s : \tau_r}{\Gamma \vdash v\ \texttt{=}\ e\ \texttt{↲}\ s : \tau_r}\ \text{(Assignment)}$$

$$\frac{\Gamma \vdash e : \tau \qquad \Gamma, v : \tau \vdash s : \tau_r}{\Gamma \vdash v\ \texttt{:}\ \tau\ \texttt{=}\ e\ \texttt{↲}\ s : \tau_r}\ \text{(Annotated Assignment)} \qquad \frac{\Gamma \vdash v \otimes e : \tau}{\Gamma \vdash v \ominus e : \top}\ \text{(Augmented Assignment)}$$

$\otimes \equiv$ *matching binary operator for* $\ominus$

$$\frac{\Gamma \vdash e : \texttt{bool} \qquad \Gamma \vdash s : \tau_r}{\Gamma \vdash \texttt{if}\ e\texttt{:}\ \texttt{↲}\ s : \tau_r}\ \text{(If)} \qquad \frac{\Gamma \vdash e : \texttt{bool} \qquad \Gamma \vdash s_1 : \tau_r \qquad \Gamma \vdash s_2 : \tau_r}{\Gamma \vdash \texttt{if}\ e\texttt{:}\ \texttt{↲}\ s_1\ \texttt{↲}\ \texttt{else:}\ \texttt{↲}\ s_2 : \tau_r}\ \text{(If Else)}$$

$$\frac{\Gamma, v : \texttt{None} \vdash s_1 : \tau_r \qquad \Gamma, v : \tau \vdash s_2 : \tau_r}{\Gamma, v : \texttt{Optional}[\tau] \vdash \texttt{if}\ v\ \texttt{is None:}\ \texttt{↲}\ s_1\ \texttt{↲}\ \texttt{else:}\ \texttt{↲}\ s_2 : \tau_r}\ \text{(Optional disambiguation)}$$

$$\frac{\Gamma \vdash e : \texttt{bool} \qquad \Gamma \vdash s : \tau_r}{\Gamma \vdash \texttt{while}\ e\texttt{:}\ \texttt{↲}\ s : \tau_r}\ \text{(While Loop)} \qquad \frac{}{\Gamma \vdash \texttt{break} : \top}\ \text{(Loop Break)} \qquad \frac{}{\Gamma \vdash \texttt{continue} : \top}\ \text{(Loop Continue)}$$

$$\frac{\Gamma \vdash e : \texttt{list}[\tau] \qquad \Gamma, v : \tau \vdash s : \tau_r}{\Gamma \vdash \texttt{for}\ v\ \texttt{in}\ e\texttt{:}\ \texttt{↲}\ s : \tau_r}\ \text{(Array Loop)} \qquad \frac{\Gamma \vdash e : \texttt{int} \qquad \Gamma, v : \texttt{int} \vdash s : \tau_r}{\Gamma \vdash \texttt{for}\ v\ \texttt{in range}(e)\texttt{:}\ \texttt{↲}\ s : \tau_r}\ \text{(Onesided Range Loop)}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int} \qquad \Gamma, v : \texttt{int} \vdash s : \tau_r}{\Gamma \vdash \texttt{for}\ v\ \texttt{in range}(e_1\texttt{,}\ e_2)\texttt{:}\ \texttt{↲}\ s : \tau_r}\ \text{(Twosided Range Loop)}$$

(d) Statements

$$\frac{\Gamma, v : \texttt{None} \vdash e_1 : \tau_r \qquad \Gamma, v : \tau \vdash e_2 : \tau_r}{\Gamma, v : \texttt{Optional}[\tau] \vdash e_1\ \texttt{if}\ v\ \texttt{is None else}\ e_2 : \tau_r}\ \text{(“}\textit{inline}\text{” Optional disambiguation)} \qquad \frac{\Gamma \vdash e_1 : \tau[] \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1[e_2] : \tau}\ \text{(Array Subscript)}$$

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \texttt{bool} \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash e_1\ \texttt{if}\ e_2\ \texttt{else}\ e_3 : \tau}\ \text{(Conditional Expression)} \qquad \frac{\Gamma \vdash e_1 : \texttt{list}[\tau] \qquad \Gamma \vdash e_2 : \texttt{int} \qquad \Gamma \vdash e_3 : \texttt{int}}{\Gamma \vdash e_1[e_2\texttt{:}e_3] : \texttt{list}[\tau]}\ \text{(Array Slice)}$$

$$\otimes \in \{+, -, *, /, \%, **\} \quad \frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \otimes e_2 : \texttt{int}}\ \text{(Integer BinOp)} \qquad \otimes \in \{<<, >>, \&, |, \hat{}\} \quad \frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \otimes e_2 : \texttt{int}}\ \text{(Bitwise BinOp)} \qquad \oslash \in \{\sim\} \quad \frac{\Gamma \vdash e : \texttt{int}}{\Gamma \vdash \oslash e : \texttt{int}}\ \text{(Bitwise UnOp)}$$

$$\otimes \in \{+, -, *, /, \%, **\} \quad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \tau_1 <: \texttt{float} \qquad \Gamma \vdash e_2 : \tau_2 \qquad \tau_2 <: \texttt{float}}{\Gamma \vdash e_1 \otimes e_2 : \texttt{float}}\ \text{(Float BinOp)} \qquad \oslash \in \{+, -\} \quad \frac{\Gamma \vdash e : \tau \qquad \tau <: \texttt{float}}{\Gamma \vdash \oslash e : \tau}\ \text{(Float UnOp)}$$

$$\otimes \in \{<, <=, ==, !=, >=, >\} \quad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \tau_1 <: \texttt{float} \qquad \Gamma \vdash e_2 : \tau_2 \qquad \tau_2 <: \texttt{float}}{\Gamma \vdash e_1 \otimes e_2 : \texttt{bool}}\ \text{(Float Comparison)}$$

$$\otimes \in \{\texttt{and}, \texttt{or}\} \quad \frac{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma \vdash e_2 : \texttt{bool}}{\Gamma \vdash e_1 \otimes e_2 : \texttt{bool}}\ \text{(Boolean BinOp)} \qquad \oslash \in \{\texttt{not}\} \quad \frac{\Gamma \vdash e : \texttt{bool}}{\Gamma \vdash \oslash e : \texttt{bool}}\ \text{(Boolean UnOp)} \qquad \otimes \in \{<, <=, ==, !=, >=, >\} \quad \frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 \otimes e_2 : \texttt{bool}}\ \text{(Generic Comparison)}$$

$$\frac{\Gamma \vdash e_1 : \texttt{str} \qquad \Gamma \vdash e_2 : \texttt{str}}{\Gamma \vdash e_1 + e_2 : \texttt{str}}\ \text{(String Concatenation)} \qquad \frac{\Gamma \vdash e_1 : \texttt{list}[\tau] \qquad \Gamma \vdash e_2 : \texttt{list}[\tau]}{\Gamma \vdash e_1 + e_2 : \texttt{list}[\tau]}\ \text{(List Concatenation)}$$

(e) Operators

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \tau_1 <: \texttt{float} \qquad \Gamma \vdash e_2 : \tau_2 \qquad \tau_2 <: \texttt{float}}{\Gamma \vdash \texttt{copysign}(e_1, e_2) : \tau_1}\ (\textsc{Builtin } \texttt{copysign})$$

$$\frac{f \in \{\texttt{min}, \texttt{max}\} \qquad \Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau \qquad \tau <: \texttt{float}}{\Gamma \vdash f(e_1, e_2) : \tau}\ (\textsc{Builtins } \texttt{min}, \texttt{max})$$

$$\frac{\Gamma \vdash e : \texttt{list}[\tau]}{\Gamma \vdash \texttt{len}(e) : \texttt{int}}\ (\textsc{Builtin } \texttt{len}(\texttt{list}))$$

$$\frac{f \in \{\texttt{ceil}, \texttt{floor}\} \qquad \Gamma \vdash e : \tau \qquad \tau <: \texttt{float}}{\Gamma \vdash f(e) : \texttt{int}}\ (\textsc{Builtins } \texttt{ceil}, \texttt{floor})$$

$$\frac{\Gamma \vdash e : \texttt{str}}{\Gamma \vdash \texttt{len}(e) : \texttt{int}}\ (\textsc{Builtin } \texttt{len}(\texttt{str}))$$

$$\frac{\Gamma \vdash e : \tau \qquad \tau <: \texttt{float}}{\Gamma \vdash \texttt{abs}(e) : \tau}\ (\textsc{Builtin } \texttt{abs})$$

$$\frac{}{\Gamma \vdash \texttt{random}() : \texttt{float}}\ (\textsc{Builtin } \texttt{random})$$

$$\frac{\Gamma \vdash e : \tau \qquad \tau <: \texttt{float}}{\Gamma \vdash \texttt{sqrt}(e) : \texttt{float}}\ (\textsc{Builtin } \texttt{sqrt})$$

$$\frac{\Gamma \vdash e : \texttt{list}[\tau]}{\Gamma, v : \texttt{list}[\tau] \vdash v.\texttt{extend}(e) : \top}\ (\textsc{List Concat})$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma, v : \texttt{list}[\tau] \vdash v.\texttt{append}(e) : \top}\ (\textsc{List Append})$$

$$\frac{}{\Gamma, v : \texttt{list}[\tau] \vdash v.\texttt{pop}() : \tau}\ (\textsc{List Pop})$$

(f) Functions

$$\frac{}{\tau <: \tau}\ (\text{“}\textit{ANYTHING}\text{”} \textsc{ is itself})$$

$$\frac{}{\bot <: \tau}\ (\bot \textsc{ is “}\textit{ANYTHING}\text{”})$$

$$\frac{}{\tau <: \top}\ (\text{“}\textit{ANYTHING}\text{”} \textsc{ is } \top)$$

$$\frac{}{\texttt{int} <: \texttt{float}}\ (\texttt{int} \textsc{ Subtype Of } \texttt{float})$$

$$\frac{\tau_1 <: \tau_2}{\tau_1 <: \texttt{Optional}[\tau_2]}\ (\tau_1 \textsc{ Subtype Of } \texttt{Optional}[\tau_2])$$

(g) Subtyping

Figure 2.3: Typing framework for ByePy

2.2 CFG Grammar

Aside from the source grammar and its semantics, we also need to define our compilation target — *i.e.*, what we want to compile to — that being a CFG representation. As the name implies, this representation focuses on making control flow explicit through a graph representation. To achieve this, we will represent all control flow through **GOTO**s — the *edges* of the CFG — and labels κ associated with specific code-blocks — the *nodes* of the CFG.

$b \coloneqq \kappa$ **:** s**;**	*labelled block*
$s \coloneqq v \leftarrow a$**;**	*statement*
IF v **THEN GOTO** κ **ELSE GOTO** κ**;**	
GOTO κ**;**	
RETURN a**;**	
RETURN;	
s**;** s	*statement sequence*
$a \coloneqq$ (⟨*scalar SQL expression*⟩)	*embeded SQL*
$v \coloneqq$ ⟨*identifier*⟩	*variable/procedure name*
$\kappa \coloneqq$ ⟨*block label*⟩	*jump target for* ***GOTO***

Figure 2.4: **GOTO**-based imperative intermediate form.

Note that this grammar *only* specifies the necessities for turing-complete control flow — *i.e.*, conditional branching. All other computation is compiled directly to scalar SQL expressions. To keep things terse, I have left the grammar of SQL expressions out of this thesis. For the interested reader, I recommend[1] diving into the SQL documents developed by the *International Organization for Standardization* (ISO) [3].

2.3 Inference Rules

With all of the groundwork for the compilation out of the way, we can finally get to the actual "*meat*" of this thesis. The ByePy compiler is defined through inference rules that transform a given snippet of ByePy code into an equivalent CFG representation in a well defined manner. To accomplish this, some complexities are needed to represent produced CFG such that rules can properly reference labels produced by other rules. To make things easier, we will take inspiration from [24] and split the translation into four inference operators:(1) $\mapsto_\kappa$, defines the translation of ByePy *statements* to CFG, (2) $\Mapsto_\kappa$, defines the translation of ByePy *expressions* to SQL, (3) $\wp_\kappa$, defines the translation of a *sequence* of ByePy *statements*, and (4) $\rightsquigarrow_\kappa$, is a little helper to generate the proper blocks and labels for loops. Notice that all operators are indexed by κ, meaning that the translations rules are all defined with the current block labeled by κ as a starting point.

As an example of reading these rules, have a look at the following dummy rule:

$$\Gamma \vdash (stmt, s) \mapsto_\kappa (\kappa', s').$$

This reads "(1) translate *stmt* to CFG, (2) starting out inside the block labeled by κ, with (3) a set of known block definition s, and (4) a set of known loop entry and exit labels Γ, (5) producing an extended set of block definitions s', and (6) a potentially new label κ'".

$$\frac{(e^{\text{Py}}, s) \Mapsto_\kappa (e^{\text{SQL}}, s_1)}{\Gamma \vdash (e^{\text{Py}}, s) \mapsto_\kappa (\kappa, s_1)}\ \text{(Bare Expression)} \qquad \frac{(e^{\text{Py}}, s) \Mapsto_\kappa (e^{\text{SQL}}, s_1) \qquad s_2 \equiv s_1 +_\kappa [\text{v} \leftarrow e^{\text{SQL}}\ \texttt{;}]}{\Gamma \vdash (v\ \texttt{=}\ e^{\text{Py}}, s) \mapsto_\kappa (\kappa, s_2)}\ \text{(Assignment)}$$

$$\frac{\Gamma \vdash (v\ \texttt{=}\ e^{\text{Py}}, s) \mapsto_\kappa (\kappa, s_1)}{\Gamma \vdash (v\texttt{:}\ \tau\ \texttt{=}\ e^{\text{Py}}, s) \mapsto_\kappa (\kappa, s_1)}\ \text{(Annotated Assignment)} \qquad \frac{\otimes \equiv \textit{matching binary operator for } \ominus \qquad \Gamma \vdash (v\ \texttt{=}\ v \otimes e^{\text{Py}}, s) \mapsto_\kappa (\kappa, s_1)}{\Gamma \vdash (v \ominus e^{\text{Py}}, s) \mapsto_\kappa (\kappa, s_1)}\ \text{(Augmented Assignment)}$$

$$\frac{s_1 \equiv s +_\kappa [\texttt{RETURN;}]}{\Gamma \vdash (\texttt{return}, s) \mapsto_\kappa (\kappa, s_1)}\ \text{(Bare Return)} \qquad \frac{(e^{\text{Py}}, s) \Mapsto_\kappa (e^{\text{SQL}}, s_1) \qquad s_2 \equiv s +_\kappa [\texttt{RETURN}\ e^{\text{SQL}}\texttt{;}]}{\Gamma \vdash (\texttt{return}\ e^{\text{Py}}, s) \mapsto_\kappa (\kappa, s_2)}\ \text{(Return)}$$

(a) Simple Statements

[1] Well, not really, but those are the official documents. For a terser — let us say more readable version — have a look at PostgreSQL's documentation [33].

$$\frac{\Gamma \vdash (\texttt{if}\ e^{\text{Py}}\texttt{:}\ \hookleftarrow\ stmts\ \hookleftarrow\ \texttt{else:}\ \hookleftarrow\ \varepsilon, s) \mapsto_\kappa (\kappa_1, s_1)}{\Gamma \vdash (\texttt{if}\ e^{\text{Py}}\texttt{:}\ \hookleftarrow\ stmts, s) \mapsto_\kappa (\kappa_1, s_1)}\ (\textsc{If})$$

$$\frac{\begin{array}{c}\kappa_{then}, \kappa_{else}, \kappa_{meet} \equiv \textit{new block labels} \qquad p \equiv \textit{new var} \\ (e^{\text{Py}}, s) \rightrightarrows_\kappa (e^{\text{SQL}}, s_1) \qquad \Gamma \vdash (stmts_1, s_1)\ \wr_{\kappa_{then}}\ (\kappa_1, s_2) \qquad \Gamma \vdash (stmts_2, s_2)\ \wr_{\kappa_{else}}\ (\kappa_2, s_3) \\ b \equiv [p \leftarrow e^{\text{SQL}}\texttt{;}, \texttt{IF}\ p\ \texttt{THEN GOTO}\ \kappa_{then}\ \texttt{ELSE GOTO}\ \kappa_{else}\ \texttt{;}] \\ s_4 \equiv s_3 +_\kappa b +_{\kappa_1} [\texttt{GOTO}\ \kappa_{meet}\ \texttt{;}] +_{\kappa_2} [\texttt{GOTO}\ \kappa_{meet}\ \texttt{;}]\end{array}}{\Gamma \vdash (\texttt{if}\ e^{\text{Py}}\texttt{:}\ \hookleftarrow\ stmts_1\ \hookleftarrow\ \texttt{else:}\ \hookleftarrow\ stmts_2, s) \mapsto_\kappa (\kappa_1, s_1)}\ (\textsc{If Else})$$

(b) Branches

$$\frac{\begin{array}{c}\kappa_{init}, \kappa_{head}, \kappa_{body}, \kappa_{end} \equiv \textit{new block labels} \\ \Gamma, \langle \kappa_{head}, \kappa_{end} \rangle \vdash (stmts, s)\ \wr_{\kappa_{body}}\ (\kappa_1, s_1) \qquad s_2 \equiv s_1 +_\kappa [\texttt{GOTO}\ \kappa_{init}\ \texttt{;}]\end{array}}{\Gamma \vdash (stmts, s) \rightsquigarrow_\kappa ((\kappa_{init}, \kappa_{head}, \kappa_{body}, \kappa_{end}), \kappa_1, s_2)}\ (\textsc{Iter})$$

$$\frac{\begin{array}{c}\Gamma \vdash (stmts, s) \rightsquigarrow_\kappa ((\kappa_{init}, \kappa_{head}, \kappa_{body}, \kappa_{end}), \kappa_1, s_1) \\ s_2 \equiv s_1 +_{\kappa_{init}} [\texttt{GOTO}\ \kappa_{head}\ \texttt{;}] +_{\kappa_{head}} [\texttt{GOTO}\ \kappa_{body}\ \texttt{;}] +_{\kappa_1} [\texttt{GOTO}\ \kappa_{head}\ \texttt{;}]\end{array}}{\Gamma \vdash (\texttt{while True:}\ \hookleftarrow\ stmts, s) \mapsto_\kappa (\kappa_{end}, s_2)}\ (\textsc{Simple Loop})$$

$$\frac{\begin{array}{c}(e^{\text{Py}}, s) \rightrightarrows_\kappa (e^{\text{SQL}}, s_1) \\ \Gamma \vdash (stmts, s_1) \rightsquigarrow_\kappa ((\kappa_{init}, \kappa_{head}, \kappa_{body}, \kappa_{end}), \kappa_1, s_2) \\ b \equiv [p \leftarrow e^{\text{SQL}}\texttt{;}, \texttt{IF}\ p\ \texttt{THEN GOTO}\ \kappa_{body}\ \texttt{ELSE GOTO}\ \kappa_{end}\ \texttt{;}] \\ s_3 \equiv s_2 +_{\kappa_{init}} [\texttt{GOTO}\ \kappa_{head}\ \texttt{;}] +_{\kappa_{head}} b +_{\kappa_1} [\texttt{GOTO}\ \kappa_{head}\ \texttt{;}]\end{array}}{\Gamma \vdash (\texttt{while}\ e^{\text{Py}}\texttt{:}\ \hookleftarrow\ stmts, s) \mapsto_\kappa (\kappa_{end}, s_3)}\ (\textsc{While Loop})$$

$$\frac{\begin{array}{c}(e^{\text{Py}}, s) \rightrightarrows_\kappa (e^{\text{SQL}}, s_1) \\ \Gamma \vdash (stmts, s_1) \rightsquigarrow_\kappa ((\kappa_{init}, \kappa_{head}, \kappa_{body}, \kappa_{end}), \kappa_1, s_2) \qquad p, v_1 \equiv \textit{new vars} \\ b_0 \equiv [v \leftarrow \texttt{0}\texttt{;}, \texttt{GOTO}\ \kappa_{head}\ \texttt{;}] \\ b_1 \equiv [v_1 \leftarrow e^{\text{SQL}}\texttt{;}, p \leftarrow v\ \texttt{<=}\ v_1\texttt{;}, \texttt{IF}\ p\ \texttt{THEN GOTO}\ \kappa_{body}\ \texttt{ELSE GOTO}\ \kappa_{end}\ \texttt{;}] \\ b_2 \equiv [v \leftarrow v\ \texttt{+1}\texttt{;}, \texttt{GOTO}\ \kappa_{head}\ \texttt{;}] \\ s_3 \equiv s_2 +_{\kappa_{init}} b_0 +_{\kappa_{head}} b_1 +_{\kappa_1} b_2\end{array}}{\Gamma \vdash (\texttt{for}\ v\ \texttt{in range(}\ e^{\text{Py}}\ \texttt{):}\ \hookleftarrow\ stmts, s) \mapsto_\kappa (\kappa_{end}, s_3)}\ (\textsc{Onesided Range Loop})$$

$$\frac{\begin{array}{c}(e_1^{\text{Py}}, s) \rightrightarrows_\kappa (e_1^{\text{SQL}}, s_1) \qquad (e_2^{\text{Py}}, s_1) \rightrightarrows_\kappa (e_2^{\text{SQL}}, s_2) \\ \Gamma \vdash (stmts, s_2) \rightsquigarrow_\kappa ((\kappa_{init}, \kappa_{head}, \kappa_{body}, \kappa_{end}), \kappa_1, s_3) \qquad p, v_1 \equiv \textit{new vars} \\ b_0 \equiv [v \leftarrow e_1^{\text{SQL}}\texttt{;}, \texttt{GOTO}\ \kappa_{head}\ \texttt{;}] \\ b_1 \equiv [v_1 \leftarrow e_2^{\text{SQL}}\texttt{;}, p \leftarrow v\ \texttt{<=}\ v_1\texttt{;}, \texttt{IF}\ p\ \texttt{THEN GOTO}\ \kappa_{body}\ \texttt{ELSE GOTO}\ \kappa_{end}\ \texttt{;}] \\ b_2 \equiv [v \leftarrow v + 1\texttt{;}, \texttt{GOTO}\ \kappa_{head}\ \texttt{;}] \\ s_4 \equiv s_2 +_{\kappa_{init}} b_0 +_{\kappa_{head}} b_1 +_{\kappa_1} b_2\end{array}}{\Gamma \vdash (\texttt{for}\ v\ \texttt{in range(}\ e_1^{\text{Py}}\texttt{,}\ e_2^{\text{Py}}\ \texttt{):}\ \hookleftarrow\ stmts, s) \mapsto_\kappa (\kappa_{end}, s_4)}\ (\textsc{Twosided Range Loop})$$

$$\frac{\Gamma \vdash (\texttt{a = e}\ \hookleftarrow\ \texttt{for i in range(len(a)):}\ \hookleftarrow\ v\ \texttt{= a[i]}\ \hookleftarrow\ stmts, s) \mapsto_\kappa (\kappa_1, s_1)}{\Gamma \vdash (\texttt{for}\ v\ \texttt{in e:}\ \hookleftarrow\ stmts, s) \mapsto_\kappa (\kappa_1, s_1)}\ (\textsc{Array Loop})$$

(c) Loops

$$\frac{s_1 \equiv s +_\kappa [\texttt{GOTO}\ \kappa_{end}\ \texttt{;}]}{\Gamma, \langle \kappa_{head}, \kappa_{end} \rangle \vdash (\texttt{break}, s) \mapsto_\kappa (\kappa, s_1)}\ \text{(Loop Break)} \qquad \frac{s_1 \equiv s +_\kappa [\texttt{GOTO}\ \kappa_{head}\ \texttt{;}]}{\Gamma, \langle \kappa_{head}, \kappa_{end} \rangle \vdash (\texttt{continue}, s) \mapsto_\kappa (\kappa, s_1)}\ \text{(Loop Continue)}$$

(d) Loop Control

$$\frac{}{\Gamma \vdash (\varepsilon, s) \fatsemi_\kappa (\kappa, s)}\ \text{(Empty Sequence)} \qquad \frac{\Gamma \vdash (stmt, s) \mapsto_\kappa (\kappa_1, s_1) \qquad \Gamma \vdash (stmts, s_1) \fatsemi_{\kappa_1} (\kappa_2, s_2)}{\Gamma \vdash (stmt\ \lrcorner\ stmts_1, s) \fatsemi_\kappa (\kappa_2, s_2)}\ \text{(Sequence)}$$

(e) Statement Sequences

$$\frac{}{(v^{\text{Py}}, s) \Mapsto_\kappa (v^{\text{SQL}}, s)}\ \text{(Variable)} \qquad \frac{}{(\langle atomic\ literal \rangle^{\text{Py}}, s) \Mapsto_\kappa (\langle atomic\ literal \rangle^{\text{SQL}}, s)}\ \text{(Atomic Literal)}$$

$$\frac{}{([\ \varepsilon\ ], s) \Mapsto_\kappa (\texttt{ARRAY}[\ \varepsilon\ ], s)}\ \text{(Empty List Literal)} \qquad \frac{(e^{\text{Py}}, s) \Mapsto_\kappa (e^{\text{SQL}}, s_1) \qquad ([\ rest^{\text{Py}}\ ], s_1) \Mapsto_\kappa (\texttt{ARRAY}[\ rest^{\text{SQL}}\ ], s_2)}{([\ e^{\text{Py}}\ \texttt{,}\ rest^{\text{Py}}\ ], s) \Mapsto_\kappa (\texttt{ARRAY}[\ e^{\text{SQL}}\ \texttt{,}\ rest^{\text{SQL}}\ ], s_2)}\ \text{(List Literal)}$$

(f) Literals

$$\frac{\tau^{\text{SQL}} \equiv \textit{matching SQL composite type for } \tau^{\text{Py}} \qquad ([args^{\text{Py}}], s) \Mapsto_\kappa (\texttt{ARRAY}[args^{\text{SQL}}], s_1)}{(\tau^{\text{Py}}(args^{\text{Py}}), s) \Mapsto_\kappa ((args^{\text{SQL}})\texttt{::}\tau^{\text{Py}}, s_1)}\ \text{(Composite Type Constructor)} \qquad \frac{(e^{\text{Py}}, s) \Mapsto_\kappa (e^{\text{SQL}}, s_1)}{(e^{\text{Py}}\texttt{.v}, s) \Mapsto_\kappa (e^{\text{SQL}}\texttt{.v}, s_1)}\ \text{(Attribute Access)}$$

(g) Composite Type Semantics

$$\frac{}{(\texttt{SQL}(\texttt{Q}), s) \Mapsto_\kappa (\texttt{Q}, s)}\ \text{(Non-parameterised Query)} \qquad \frac{([args^{\text{Py}}], s) \Mapsto_\kappa (\texttt{ARRAY}[args^{\text{SQL}}], s_1)}{(\texttt{SQL}(\texttt{"Q"}\texttt{,}\ [args^{\text{Py}}]), s) \Mapsto_\kappa (\texttt{Q}[args^{\text{SQL}}], s_1)}\ \text{(Parameterised Query)}$$

(h) Queries

$$\frac{\otimes^{\text{SQL}} \equiv \textit{matching SQL operator for } \otimes^{\text{Py}} \qquad (e_1^{\text{Py}}, s) \Mapsto_\kappa (e_1^{\text{SQL}}, s_1) \qquad (e_2^{\text{Py}}, s_1) \Mapsto_\kappa (e_2^{\text{SQL}}, s_2)}{(e_1^{\text{Py}} \otimes^{\text{Py}} e_2^{\text{Py}}, s) \Mapsto_\kappa (e_1^{\text{SQL}} \otimes^{\text{SQL}} e_2^{\text{SQL}}, s_1)}\ \text{(Binary Operator)}$$

$$\frac{\oslash^{\text{SQL}} \equiv \textit{matching SQL operator for } \oslash^{\text{Py}} \qquad (e^{\text{Py}}, s) \Mapsto_\kappa (e^{\text{SQL}}, s_1)}{(\oslash^{\text{Py}} e^{\text{Py}}, s) \Mapsto_\kappa (\oslash^{\text{SQL}} e^{\text{SQL}}, s_1)}\ \text{(Unary Operator)}$$

$$\frac{(e^{\text{Py}}, s) \Mapsto_\kappa (e^{\text{SQL}}, s_1)}{(e^{\text{Py}}\ \texttt{is not None}, s) \Mapsto_\kappa (e^{\text{SQL}}\ \texttt{IS NOT NULL}, s_1)}\ \text{(Not Null Check)} \qquad \frac{(e^{\text{Py}}, s) \Mapsto_\kappa (e^{\text{SQL}}, s_1)}{(e^{\text{Py}}\ \texttt{is None}, s) \Mapsto_\kappa (e^{\text{SQL}}\ \texttt{IS NULL}, s_1)}\ \text{(Null Check)}$$

$$\frac{(e_1^{\text{Py}}, s) \Mapsto_\kappa (e_1^{\text{SQL}}, s_1) \qquad (e_2^{\text{Py}}, s_1) \Mapsto_\kappa (e_2^{\text{SQL}}, s_2) \qquad (e_3^{\text{Py}}, s_2) \Mapsto_\kappa (e_3^{\text{SQL}}, s_3)}{(e_1^{\text{Py}}\ \texttt{if}\ e_2^{\text{Py}}\ \texttt{else}\ e_3^{\text{Py}}, s) \Mapsto_\kappa (\texttt{CASE WHEN}\ e_2^{\text{SQL}}\ \texttt{THEN}\ e_1^{\text{SQL}}\ \texttt{ELSE}\ e_3^{\text{SQL}}\ \texttt{END}, s_3)}\ \text{(Conditional Expression)}$$

(i) Operators

$$\frac{(e_1^{Py}, s) \mapsto_\kappa (e_1^{SQL}, s_1) \qquad (e_2^{Py}, s_1) \mapsto_\kappa (e_2^{SQL}, s_2)}{(e_1^{Py}[e_2^{Py}], s) \mapsto_\kappa (e_1^{SQL}[e_2^{SQL}+1], s_2)} \text{ (Array Subscript)}$$

$$\frac{(e_1^{Py}, s) \mapsto_\kappa (e_1^{SQL}, s_1) \qquad (e_2^{Py}, s_1) \mapsto_\kappa (e_2^{SQL}, s_2)}{(e_1^{Py}[e_2^{Py}\texttt{:}], s) \mapsto_\kappa (e_1^{SQL}[e_2^{SQL}\texttt{+1:}], s_2)} \text{ (Lower Bounded Array Slice)}$$

$$\frac{(e_1^{Py}, s) \mapsto_\kappa (e_1^{SQL}, s_1) \qquad (e_2^{Py}, s_1) \mapsto_\kappa (e_2^{SQL}, s_2)}{(e_1^{Py}[\texttt{:}e_2^{Py}], s) \mapsto_\kappa (e_1^{SQL}[\texttt{:}e_2^{SQL}], s_2)} \text{ (Upper Bounded Array Slice)}$$

$$\frac{(e_1^{Py}, s) \mapsto_\kappa (e_1^{SQL}, s_1) \qquad (e_2^{Py}, s_1) \mapsto_\kappa (e_2^{SQL}, s_2) \qquad (e_3^{Py}, s_2) \mapsto_\kappa (e_3^{SQL}, s_3)}{(e_1^{Py}[e_2^{Py}\texttt{:}e_3^{Py}], s) \mapsto_\kappa (e_1^{SQL}[e_2^{SQL}\texttt{+1:}e_3^{SQL}], s_3)} \text{ (Fully Bounded Array Slice)}$$

(j) Subscript Operators

$$\frac{(e_1^{Py}, s) \mapsto_\kappa (e_1^{SQL}, s_1) \qquad (e_2^{Py}, s_1) \mapsto_\kappa (e_2^{SQL}, s_2)}{(\texttt{copysign}(e_1^{Py}\texttt{,}e_2^{Py}), s) \mapsto (\texttt{ABS}(e_1^{SQL})\texttt{*}\ \texttt{SIGN}(e_2^{SQL}), s_2)} \text{ (Builtin Function: } \texttt{copysign})$$

$$\frac{(e^{Py}, s) \mapsto_\kappa (e^{SQL}, s_1)}{(\texttt{len}(e^{Py}), s) \mapsto_\kappa (\texttt{CARDINALITY}(e^{SQL}), s_1)} \text{ (Builtin Function: } \texttt{len}) \qquad \frac{}{(\texttt{random}(), s) \mapsto_\kappa (\texttt{RANDOM}(), s)} \text{ (Builtin Function: } \texttt{random})$$

$$\frac{(e^{Py}, s) \mapsto_\kappa (e^{SQL}, s_1)}{(\texttt{abs}(e^{Py}), s) \mapsto_\kappa (\texttt{ABS}(e^{SQL}), s_1)} \text{ (Builtin Function: } \texttt{abs}) \qquad \frac{f^{SQL} \equiv \textit{matching SQL function for } f^{Py} \qquad ([args^{Py}], s) \mapsto_\kappa (\texttt{ARRAY}[args^{SQL}], s_1)}{(f^{Py}(args^{Py}), s) \mapsto_\kappa (f^{SQL}(args^{SQL}), s_1)} \text{ (Generic Builtin Function)}$$

(k) Functions

$$\frac{(e^{Py}, s) \mapsto_\kappa (e^{SQL}, s_1) \qquad s_2 \equiv s_1 +_\kappa [v \leftarrow v\ \texttt{||}\ e^{SQL}\texttt{;}]}{\Gamma \vdash (v\texttt{.}\texttt{append}(e^{Py}), s) \mapsto_\kappa (\kappa, s_2)} \text{ (Stateful List Append)}$$

$$\frac{(e^{Py}, s) \mapsto_\kappa (e^{SQL}, s_1) \qquad s_2 \equiv s_1 +_\kappa [v \leftarrow v\ \texttt{||}\ e^{SQL}\texttt{;}]}{\Gamma \vdash (v\texttt{.}\texttt{extend}(e^{Py}), s) \mapsto_\kappa (\kappa, s_2)} \text{ (Stateful List Extend)}$$

$$\frac{p \equiv \textit{new var} \qquad s_1 \equiv s +_\kappa [p \leftarrow v[1]\texttt{;}, v \leftarrow v[2\texttt{:}]\texttt{;}]}{(v\texttt{.}\texttt{pop}(), s) \mapsto_\kappa (p, s_1)} \text{ (Statefule List Pop)}$$

(l) *Stateful* Functions

2.4 Backend

With the first step in Figure 2.1 sufficiently covered we can close this chapter out with the promised overview of the remainder of the steps. Again this will be fairly terse and by no means a complete description — for that, I highly recommend diving into the original papers by Denis Hirn et al. [21, 22, 24].

CFG to SSA. You can think of SSA as a specialized form of CFGs. As the name implies, SSA guarantees that any variable is defined only once. This — among other properties — makes it easier to reason about programs in SSA form. The main difficulty of transforming programs to this representation becomes apparent when you think of loops that mutate variables — you are not allowed to reassign an existing variable. Instead, you have to do two things. First, you need to assign each variable a unique *version* — for example, $x \leftarrow 1; x \leftarrow x + 1$ would become $x_0 \leftarrow 1; x_1 \leftarrow x_0 + 1$. Second, every time a block can be entered from multiple other blocks, you need to place a Φ-function for all variables whose version is now ambiguous — *i.e.*, that could have been mutated in any of the code paths.

SSA to ANF. ANF is a functional representation that is equivalent to SSA [34, 35]. What SSA represents as labelled blocks and `GOTO`, ANF represents through functions and tail calls. Transforming between these two representations is not trivial, but luckily there is extensive work that we rest easy upon; the specific compilation pipeline described by Hirn et al. uses the transformations shown in [35] to transform between the two.

Trampolining ANF. Trampolined style [36] code is a specific style of functional code in which all programs only have two major functions a `start` function and a `step` function. The entry point into the program is — as the name implies — the `start` function, which applies the `step` function to the initial state of the program. The `step` function then handles all control flow and state mutation of the program through self-recursion. This step is pretty *meaty* and as not to *butcher* it too much, we defer the proper description to [24].

Trampolined ANF to SQL. Last but not least, we can translate the now trampolined ANF to SQL. To do so, we start by translating the `start` and `step` functions into individual SQL queries by mapping

1. chains of `LET` bindings to chains of `LATERAL` joins,
2. function calls — which all call the `step` function after trampolining — to `SELECT` statements that contain the current state, and lastly
3. conditional expressions to both conditional branches glued together with `UNION ALL` and opposing `WHERE` conditions.

Notice that we only need to translate the control flow since everything else is already in SQL! Once we have translated both the `start` and `step` functions in SQL, all that remains is to place them into a recursive CTE — as seen in Listing 2.3.

```
WITH RECURSIVE run("rec?", res, ...) AS
    ( SELECT ...   -- ← start function
        UNION ALL
      SELECT ... ) -- ←  step function
  SELECT run.res FROM run WHERE NOT run."rec?";
```

Listing 2.3: Stub of a compiled query

3

Experiments

But does this compilation yield the performance benefits it set out to achieve? Do the promises of cutting down on overhead hold? Or does this shift the problem from multiple smaller overheads to one big one? To answer these questions, let us have a look at some experiments.

To determine the efficacy of the compilation, we first need some example programs. For this, we can adapt some examples from [24]. Half of the examples use the TPC-H [37] dataset — an industry-standard data set explicitly for comparable testing on realistic data; the other half is a collection of data-heavy algorithms — see Table 3.1 for a list of all sample programs. Amongst the details listed in Table 3.1, you will also find *Cyclomatic Complexity* (CC) [38] — a measure linearly independent control flow paths through a function — and the shape of a program — *i.e.*, the constituent loops and their respective nesting.

Table 3.1: Sample Programs

	Program	TPC-H?	CC	Shape
`margin`	buy/sell TPC-H orders to maximize margin	✔	3	⌊
`savings`	supply chain optimization of TPC-H orders	✔	4	⌊
`packing`	optimal packing of TPC-H orders into containers	✔	9	⌊⌊
`force`	n-body simulation using a database resident quadtree	✘	5	⌊
`march`	marching squares with database resident geometry data	✘	5	⌊
`markov`	robot control based on Markov-chains	✘	5	⌊
`vm-collatz`	execute a database collatz conjecture resident program in a virtual machine	✘	17	⌊
`vm-padovan`	execute a database padovan sequence resident program in a virtual machine	✘	17	⌊

Comparing the execution time of the Python version against the compiled version does not show the whole picture. We also need to look at how the compiled version holds up compared to other easily accessible options for making computation database "*resident*" — PL/SQL and PL/Python. While doing so, it is also essential to ensure that all versions can exhibit their best performance — for example, including Python-based plan caching as demonstrated in the PL/Python documentation [15, 7. Database Access] or prepared statements [39], which achieves the plan caching for the ByePy code.

3.1 Setup

We run all the experiments on an experiment server provided by the Database System Research Group. The machine has two AMD EPYC™ 7402 24-Core processors [40], and 2 TB of Samsung ECC DDR4 DRAM [41]. As DBMS we use PostgreSQL— specifically, version 11.3 — as it is widely used and has support for both PL/SQL in the form of PL/pgSQL [14] and PL/Python through the `plpython3u` extension [15].

We use Python to run all experiments and to collect all measurement data — *i.e.*, through a script that calls into the database and collects statistics — see Table 3.2 for an enumeration. This is required to ensure *fairness* between all versions since ByePy style Python code is intended to run entirely outside of the database. For this, we used Python 3.8 and Psycopg2 2.9.1. To extract internal database statistics — *e.g.*, time spent planning queries *versus* executing them — we also use a PostgreSQL extension that uses PostgreSQL's extensive hook infrastructure to inject some timers at interesting points during query processing.

Table 3.2: Captured statistics

Statistic		Method
`time`	Total execution time	*via* Python
***SQL*`-calls`**	Number of calls to the ***SQL***-function	*via* Python
`exec-start`	Time spent initializing query executors	*via* PostgreSQL
`exec-run`	Time spent actually running the query executors	*via* PostgreSQL
`exec-finish`	Time spent post-processing results of the query executors	*via* PostgreSQL
`exec-end`	Time spent cleaning-up the query executors	*via* PostgreSQL
`plan-build-time`	Time spent planning queries	*via* PostgreSQL
`plan-build-count`	Number of calls to the query planner	*via* PostgreSQL
`plan-cache-time`	Time spent looking up query plans in the plan cache	*via* PostgreSQL
`plan-cache-count`	Number of query plan lookups	*via* PostgreSQL

You need to keep three things in mind when looking at the numbers later. The first is that `time` does not necessarily need to be the sum of all the other listed duration stats as it also includes network and python overhead. Further, `time` is not only not necessarily equal to the sum of all other duration metrics, it does not even have to be bigger than that sum — the database hands off results back to Python before the plan teardown is completed or sometimes even started. And, last but not least, keep in mind, that the ***SQL*`-calls`** stat is fairly boring for all execution models other than ByePy itself, since PL/Python, PL/SQL, and SQL all call into the database exactly *once* per call of their corresponding Python-function in the experimental setup.

We can parameterize our measurements over the number of *invocations* of our function and the number of *iterations* we roughly expect a given function invocation to incur. For the TPC-H-based examples, we can even go a step further and use the *Scaling Factor* (SF) of the underlying TPC-H-instance — *i.e.*, the size of the instance in `GB` — stand in for the number of invocations and use a fraction thereof as our iterations. Increasing the two respective parameters gives us a 2D result space that we can easily look at in the form of heat maps.

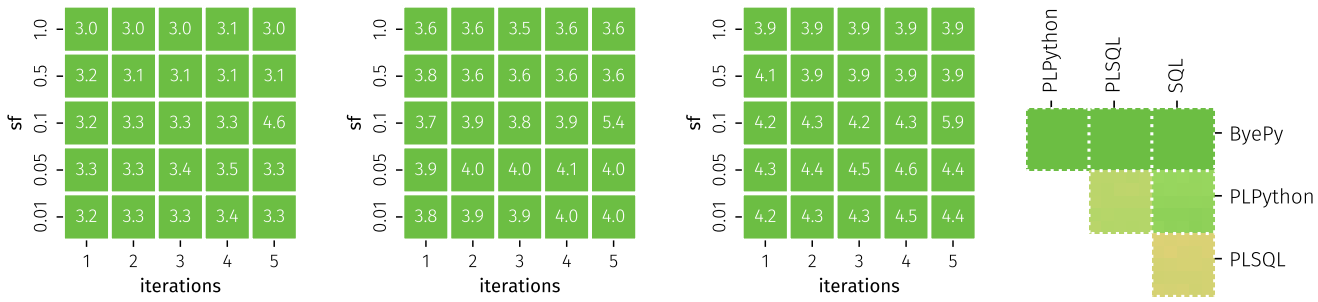
3.2 Experiments

With all the pro forma out of the way, we can finally look at some results — see Table 3.2. We can observe speedups of up to an order of magnitude when comparing the ByePy-style experiments with their compiled SQL counterparts. And we can also see that the SQL version outperforms the PL/Python and PL/SQL versions.

Table 3.3: An assorted collection of collected metrics for each example program

Program	Runtime (speedup) of SQL						# *SQL* calls
	vs. ByePy		*vs. PL/Python*		*vs. PL/SQL*		
`margin`	24%	(4.2×)	77%	(1.3×)	91%	(1.1×)	72738
`savings`	5%	(19.5×)	33%	(3.0×)	43%	(2.3×)	785440
`packing`	16%	(6.3×)	71%	(1.4×)	83%	(1.2×)	1964482
`force`	27%	(3.9×)	91%	(1.1×)	125%	(0.8×)	2681
`march`	13%	(7.6×)	83%	(1.2×)	100%	(1.0×)	8420
`markov`	39%	(2.6×)	83%	(1.2×)	91%	(1.1×)	11877
`vm-collatz`	30%	(3.3×)	111%	(0.9×)	100%	(1.0×)	2752
`vm-padovan`	12%	(8.5×)	50%	(2.0×)	63%	(1.6×)	28925

But this is not the whole story. In Table 3.3, we can see the numbers for the middlemost measurement — *i.e.*, the measurement with median invocation and iteration values. We use heat maps – like in Figure 3.1 — to get a better look at all measured speedups at the same time.



(a) PL/Python *vs.* ByePy **(b)** PL/SQL *vs.* ByePy **(c)** Compiled SQL *vs.* ByePy **(d)** Meta heat map

Figure 3.1: Speedups of all versions of `margin` over the ByePy version

If we zoom out a bit, we can easily see how all of the versions compare to one another via "*meta heat maps*" — like Figure 3.1d. In the following discussion of dilemmas some of the experiments face, we will look at these a few times to gain a birdseye view of the symptoms.

3.2.1 Not all is well that ends well

Though Table 3.3 shows a rosy picture of the experimental results there are some caveats. They become easily visible in the teased meta heat maps — have a look at Figure 3.2.

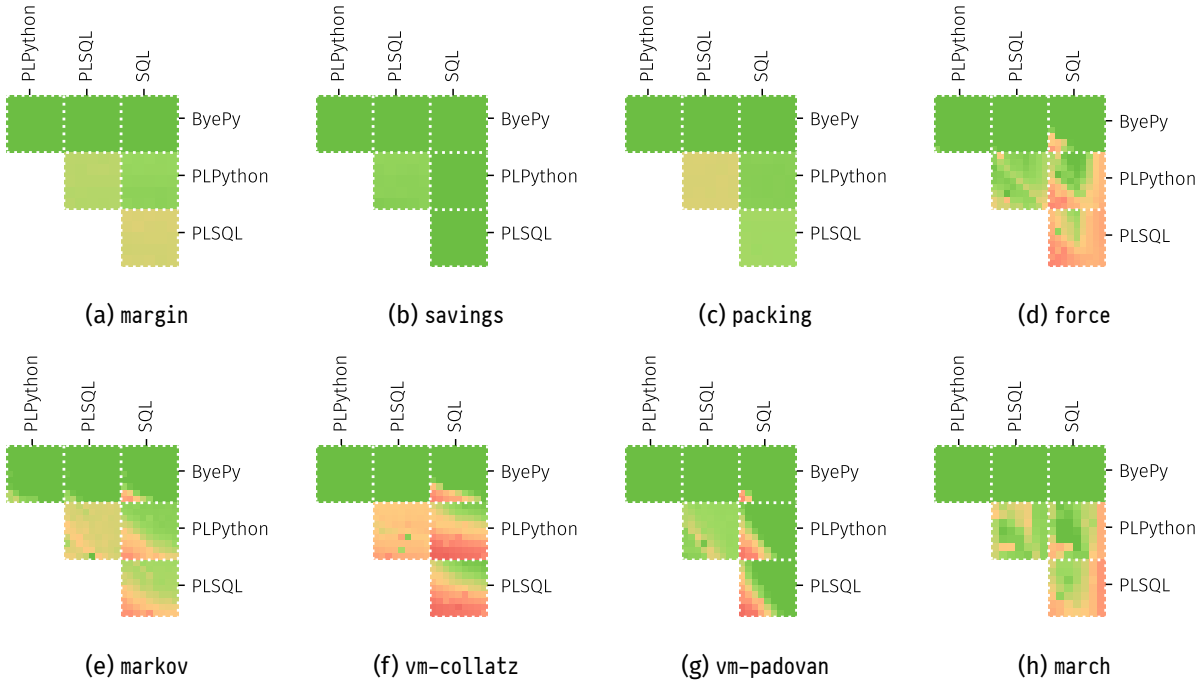


Figure 3.2: Speedup meta heat maps for all examples.

All the TPC-H examples behave exactly as expected — they show speedups across the board. But the two-dimensional representation is disingenuous, as the SF only influences the total size of a given TPC-H instance; not the number of relationships between different rows. Overall the cardinality of relationships stays roughly the same — *i.e.*, a given order will always contain around about the same number of items, no matter the SF.

The Influence of Query Overhead. In difference to the TPC-H examples, we can observe that both `markov` (Figure 3.2e) and `vm-collatz` (Figure 3.2f) appear to exhibit some difficulties with smaller inputs. These examples execute many small queries where the query's overhead dominates its runtime. The actual query runtime is still present after compilation, but the query's overhead is not. The closer we are to the top right corner of the heat maps, the more overhead accumulates, yielding big speedups upon elimination through the compilation. Proximity to the lower left corner signals the opposite, there is barely any overhead to be eliminated, and the overhead that has accumulated is smaller than the overhead provoked by the significantly more complicated compiled query.

Table 3.4: Number of executed queries in BytePy/PL/pgSQL/PL/Python programs.

Iterations	# queries	
	markov	vm-collatz
4	42	76
8	174	200
16	732	496
32	2.958	1.184
64	11.877	2.752
128	44.517	6.272
256	186.438	14.080
512	785.694	31.232
1024	3.072.609	68.608

The Cost of Memory. Another anomaly is the significant slowdown for large numbers of iterations for `force` (Figure 3.2d) and `march` (Figure 3.2h). Both of these examples use intermediate data structures that grow larger with each iteration — or at least do so for a significant portion of all iterations. Take `march` for example; it implements the marching square algorithm that determines the boundary of a 2D shape. Said boundary is accumulated in an array that needs to be replicated and potentially extended for each entry in the working table of the compiled version.

Table 3.5: Memory allocation due to `UNION` table construction.

Iterations	`UNION` table size (bytes)	
	force	march
4	1.385	5.084
8	3.325	18.000
16	8.101	68.000
32	15.000	264.000
64	34.000	1.040.000
128	64.000	4.127.000
256	107.000	16.000.000
512	154.000	64.000.000
1024	219.000	256.000.000

4 Conclusion

The translation we presented and explored in this thesis demonstrates that we can compile a subset of Python to SQL and thus *move computation close to the data*. We can even deduce the broader applicability of the compilation method presented by Hirn et al. in [24]. This compilation method can compile the code of a wide range of imperative languages to recursive SQL queries without needing to do the heavy lifting of translating imperative semantics to equivalent declarative expressions. And depending on the *distance* between the languages runtime and the database — be it virtual or physical — we can expect a significant speedup. In the case of ByePy, we saw a speedup of up to an *order of magnitude* in the experiments in Chapter 3.

But we also saw that another difficulty could arise from the typing model, aside from the impedance mismatch of language paradigms. Since Python is dynamically typed, we needed to introduce not only a specific Python grammar subset but also a stringent set of typing rules. Further, this set of typing rules needs to make sure it covers "non-value values" like `None` properly. Though this is not that big of a problem for languages that implement functor-like handling of such values, as SQL's `NULL`-value behaves like the `Maybe`-functor of languages like Haskell [42].

Another not immediately obvious point is that we can use the compilation results in a multitude of DBMSs. All that is required is support for turing complete queries — *i.e.*, `WITH RECURSIVE` or similar constructs. Though the current iteration of the backend produces `LATERAL` joins, you can replace these with `LEFT OUTER JOIN` joins — as is the case in the [24]. This already includes a wide set of DBMSs, for example, PostgreSQL, Umbra [43], HyPer [44], MySQL [45], Oracle Database [46], and Microsoft SQL Server [47] — in short, quite a few.

4.1 Future Work

Take the following presentations of future work with a grain of salt, we cannot expect all of these to work out as we hope — but we can still dream. We will start with looking at some that only affect ByePy and end with some that are transferrable to other languages.

4.1.1 Python Dictionaries

With `list`s ByePy currently supports only one of the two major container types in Python. The other container type is Python dictionaries; arguably an equally important container type. Any developer trying to write pythonic code using the ByePy syntax will find this to be a big stumbling block. To bridge this gap a bit, we could implement a limited variant of Python dictionaries using the commonly supported *JavaScript Object Notation* (JSON) container type [48, 49, 50, 51, 52]. The

ubiquity of JSON makes it a useful representation to ensure portability, but we would also need to accept it being limited to `str`-valued keys.

4.1.2 Python "Query" Comprehensions

You may know that Python has a syntactic construct that is eerily similar to SQL queries — the *comprehension*. Any system claiming to be pythonic should use comprehensions as they are one of the defining features of the Python syntax. We could even go so far as translating comprehension more efficiently as aggregate queries — see Listing 4.1.

```
list_of_numbers = [
  row.num                          # SELECT ARRAY_AGG(row.num)
  for row in table("some-table")   #   FROM "some-table" AS row
  if row.bool or row.num > 1       #  WHERE row.bool AND row.numer > 1
]

biggest_number = max(
  row.num                          # SELECT MAX(row.num)
  for row in table("some-table")   #   FROM "some-table" AS row
)
```

Listing 4.1: Examples for "Query" Comprehensions and their respective query forms

The intermediate Python developer may even know of another special type of comprehension — the *generator comprehension* — that just builds an iterator instead of materializing a comprehensions entire results. If we combine generator comprehensions with the ideas presented in Section 4.1.4 we can achieve even more pythonic code while forgoing even more SQL syntax for simple *Select-From-Where* (SFW) queries.

4.1.3 Python Comprehension Desugaring

An orthogonal approach to Section 4.1.2 is the *desuggaring* of comprehensions to their `for`-loop based counterparts — see Listing 4.2. This loop-based form can then be optimized by generic loop optimizations — like Section 4.1.5 or Section 4.1.6.

```
nums = [1,2,3,4,5,6,7,8,9,10]

# Original
list_of_numbers = [
  num ** 2
  for num in nums
]

# Desugared
list_of_numbers = []
for row in rows:
    list_of_numbers.append(row.num ** 2)
```

Listing 4.2: Example of Comprehension Desugaring

4.1.4 Iterating over Query Results

The avid Python developer — or for that point, anyone who has used imperative languages for any extended amount of time — will have noticed one massive limitation in the current state of the ByePy grammar. That is the lack of an easy way to iterate through a result set of a query. The current grammar forces you to materialize the entire result in a scalar value via array aggregation. And as you may have noticed, this is far from optimal. As of yet, this is a point of contention as "*proper*" pythonic code should use iterators more naturally.

To include iteration over query results, we need a way to represent a query result as a scalar SQL value — other than arrays, of course. For this, we could make use of SQL cursors [53, 54, 55, 56], but current efforts show some difficulties with this approach. Other than cursors, there are no options to represent "running queries" as SQL scalars; all other approaches fall back to materializing the entire query result in one way or another.

4.1.5 Map-like Loops

Map-like loops that perform some operation on each element in a container. They derive their name from the high-order function `map` that implements the same functionality by abstracting over the elementwise operation.

```
nums: list[int] = [1,2,3,4,5,6,7,8,9,10]
for idx in range(9):
    nums[idx] = num[idx] ** 2
```

Listing 4.3: Example of a Map-like Loop

Instead of translating map-like loops like other loops, they can be implemented more efficiently in SQL. As things stand, each iteration of the loop corresponds to one step of the recursive query. But there are methods in SQL to iterate through container types using standard query notation — *i.e.*, by including them in the `FROM`-list via functions that turn them into a relation. In PostgreSQL you can, for example, use the `unnest` function to turn an array into a relation and the `json_each` function to turn a JSON object into a relation of key-value pairs.

```
SELECT "nums2"
  FROM LATERAL (SELECT ARRAY[1,2,3,4,5,6,7,8,9,10]) AS "let1"("nums1")
     , LATERAL (SELECT ARRAY_AGG("num" ^ 2)
                  FROM unnest("nums1") AS _("num")) AS "let2"("nums2")
```

Listing 4.4: Example translation of Listing 4.3 to SQL

4.1.6 Fold-like Loops

Fold-like loops represent a generalization of map-like loops (Section 4.1.5) in the same way the `fold`/`reduce` function represents a generalization of the `map` function in the functional world. To keep things simple, you can assume that fold-like loops just extend map-like loops with the functionality (1) to skip/filter elements, and (2) to combine elements with their predecessors in the container.

```python
nums: list[int] = [1,2,3,4,5,6,7,8,9,10]
maximum: Optional[int] = None
for num in nums:
    if maximum is None:
        maximum = num
    else maximum < num:
        maximum = num
```

Listing 4.5: Example of a Fold-like Loop

The most common fold-like loops out in the field find parity in common aggregate functions in SQL — *e.g.*, `MIN`, `MAX`, `ARRAY_AGG`[1], *etc.* Such loops are not necessarily that nice to write by hand, but they could represent an ideal target for the desugaring presented in Section 4.1.3.

```sql
SELECT "maximum"
  FROM LATERAL (SELECT ARRAY[1,2,3,4,5,6,7,8,9,10]) AS "let1"("nums")
     , LATERAL (SELECT MAX("num")
                  FROM unnest("nums") AS _("num"))  AS "let2"("maximum")
```

Listing 4.6: Example translation of Listing 4.5 to SQL

This approach to handling fold-like loops — and in extension, map-like ones as well — is similar in style to how Gupta et al. approached general loops in their work on *Aggify* [57]. Though in contrast they generate custom aggregate functions written in C# instead of detecting and optimizing such loops in the realm of SQL.

[1] Notice that using `ARRAY_AGG` and no filtering makes a fold-like loop behave exactly like a map-like loop!

Bibliography

[1] Tim Fischer, Denis Hirn, and Torsten Grust. “Snakes on a Plan: Compiling Python Functions into Plain SQL Queries”. In: *Proceedings of the 2022 International Conference on Management of Data*. SIGMOD ’22. Philadelphia, PA, USA: Association for Computing Machinery, 2022, pp. 2389–2392. ISBN: 9781450392495. DOI: 10.1145/3514221.3520175. URL: https://doi.org/10.1145/3514221.3520175.

[2] JetBrains. *Python Developers Survey 2021*. 2021. URL: https://lp.jetbrains.com/python-developers-survey-2021/.

[3] *SQL:1999 Standard. Database Languages–SQL–Part 2: Foundation*. ISO/IEC 9075-2:1999.

[4] Martin Fowler. *Patterns of Enterprise Application Architecture*. 1st ed. Addison-Wesley Professional, 2002. ISBN: 0321127420-9780321127426.

[5] L.A. Rowe and M. Stonebraker. “The POSTGRES Data Model”. In: *Proc. VLDB*. 1987. URL: https://www.vldb.org/conf/1987/P083.PDF.

[6] Charles R. Harris et al. “Array programming with NumPy”. In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

[7] Wes McKinney. “Data Structures for Statistical Computing in Python”. In: *Proceedings of the 9th Python in Science Conference*. Ed. by Stéfan van der Walt and Jarrod Millman. 2010, pp. 51–56. URL: https://conference.scipy.org/proceedings/scipy2010/pdfs/mckinney.pdf.

[8] Bogumił Kamiński et al. *JuliaData/DataFrames.jl*. May 2022. DOI: 10.5281/zenodo.3376177. URL: https://doi.org/10.5281/zenodo.3376177.

[9] Matthew Rocklin. “Dask: Parallel Computation with Blocked algorithms and Task Scheduling”. In: *Proceedings of the 14th Python in Science Conference*. Ed. by Kathryn Huff and James Bergstra. 2015, pp. 130–136.

[10] The Apache Software Foundation. *Apache Parquet*. 2013. URL: https://parquet.apache.org/.

[11] Judegment of 16 July 2020, *Data Protection Commissioner* v *Facebook Ireland Ltd, Maximillian Schrems*, C-311/18, ECLI:EU:C:2020:559. URL: https://curia.europa.eu/juris/document/document.jsf?docid=228677&doclang=en&cid=9791227.

[12] Oracle. *Oracle Database Release 21 — PL/SQL*. 2021. URL: https://docs.oracle.com/en/database/oracle/oracle-database/21/lnpls/index.html.

[13] Microsoft. *SQL Server 2022 — T-SQL*. 2022. URL: https://docs.microsoft.com/en-us/sql/t-sql/language-reference?view=sql-server-ver16.

[14] The PostgreSQL Global Development Group. *PostgreSQL version 11 — PL/pgSQL*. 2018. URL: https://www.postgresql.org/docs/11/plpgsql.html.

[15] The PostgreSQL Global Development Group. *PostgreSQL version 11 — PL/Python*. 2018. URL: https://www.postgresql.org/docs/11/plpython.html.

[16] Martin Grund et al. "Power to the SQL People: Introducing Python UDFs in Databricks SQL". In: *Databricks Blog* (2022). URL: https://www.databricks.com/blog/2022/07/22/power-to-the-sql-people-introducing-python-udfs-in-databricks-sql.html.

[17] Snowflake inc. *Snowflake — Python UDFs*. URL: https://docs.snowflake.com/en/developer-guide/udf/python/udf-python.html.

[18] Snowflake inc. *Snowflake — Java UDFs*. URL: https://docs.snowflake.com/en/developer-guide/udf/java/udf-java.html.

[19] Snowflake inc. *Snowflake — JavaScript UDFs*. URL: https://docs.snowflake.com/en/developer-guide/udf/javascript/udf-javascript.html.

[20] Karthik Ramachandra et al. "Froid: Optimization of Imperative Programs in a Relational Database". In: *Proc. VLDB Endow.* 11.4 (Dec. 2017), pp. 432–444. ISSN: 2150-8097. DOI: 10.1145/3164135.3164140. URL: https://doi.org/10.1145/3164135.3164140.

[21] Christian Duta, Denis Hirn, and Torsten Grust. *Compiling PL/SQL Away*. 2019. DOI: 10.48550/ARXIV.1909.03291. URL: https://arxiv.org/abs/1909.03291.

[22] Denis Hirn and Torsten Grust. "PL/SQL Without the PL". In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. SIGMOD '20. Portland, OR, USA: Association for Computing Machinery, 2020, pp. 2677–2680. ISBN: 9781450367356. DOI: 10.1145/3318464.3384678. URL: https://doi.org/10.1145/3318464.3384678.

[23] Christian Duta and Torsten Grust. "Functional-Style SQL UDFs With a Capital 'F'". In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. SIGMOD '20. Portland, OR, USA: Association for Computing Machinery, 2020, pp. 1273–1287. ISBN: 9781450367356. DOI: 10.1145/3318464.3389707. URL: https://doi.org/10.1145/3318464.3389707.

[24] Denis Hirn and Torsten Grust. "One WITH RECURSIVE is Worth Many GOTOs". In: *Proceedings of the 2021 International Conference on Management of Data*. SIGMOD '21. Virtual Event, China: Association for Computing Machinery, 2021, pp. 723–735. ISBN: 9781450383431. DOI: 10.1145/3448016.3457272. URL: https://doi.org/10.1145/3448016.3457272.

[25] Tobias Burghardt, Denis Hirn, and Torsten Grust. "Functional Programming on Top of SQL Engines". In: *Practical Aspects of Declarative Languages*. Ed. by James Cheney and Simona Perri. Cham: Springer International Publishing, 2022, pp. 59–78. ISBN: 978-3-030-94479-7.

[26] Stack Overflow. *Stack Overflow Developer Survey 2022*. 2022. URL: https://insights.stackoverflow.com/survey/2022.

[27] JetBrains. *The State of Developer Ecosystem 2021*. 2021. URL: https://www.jetbrains.com/lp/devecosystem-2021/.

[28] Python Software Foundation. *Python —Duck Typing*. 2022. URL: https://docs.python.org/3.8/glossary.html#term-duck-typing.

[29] Guido van Rossum, Jukka Lehtosalo, and Łukasz Langa. *PEP 484 – Type Hints*. 2014. URL: https://peps.python.org/pep-0484/.

[30] the mypy project. *mypy*. 2014. URL: http://mypy-lang.org/.

[31] Robert Craigie. *pyright*. 2021. URL: https://pypi.org/project/pyright/.

[32] Tim Peters. *PEP 20 — The Zen of Python*. 2004. URL: https://peps.python.org/pep-0020/.

[33] The PostgreSQL Global Development Group. *PostgreSQL version 11*. 2018. URL: https://www.postgresql.org/docs/11/.

[34] A.W. Appel. "SSA is Functional Programming". In: *ACM SIGPLAN Notices* 33.4 (Apr. 1998).

[35] M. Chakravarty, G. Keller, and P. Zadarnowski. "A Functional Perspective on SSA Optimisation Algorithms". In: *Electronic Notes in Theoretical Computer Science* 82.2 (2004).

[36] S.E. Ganz, D.P. Friedman, and M. Wand. "Trampolined Style". In: *Proc. ICFP*. Paris, France, Sept. 1999.

[37] TPC. *TPC Benchmark H (Revision 3.0.0)*. 2021. URL: http://tpc.org/tpch.

[38] T.J. McCabe. "A Complexity Measure". In: *IEEE Transactions on Software Engineering* SE-2.4 (1976), pp. 308–320. DOI: 10.1109/TSE.1976.233837.

[39] The PostgreSQL Global Development Group. *PostgreSQL version 11 — **PREPARE***. 2018. URL: https://www.postgresql.org/docs/11/sql-prepare.html.

[40] AMD. *AMD EPYC™ 7402*. URL: https://www.amd.com/en/products/cpu/amd-epyc-7402.

[41] Samsung Seminconductor Global. *Samsung 64Gb ECC DDR4 DRAM*. URL: https://semiconductor.samsung.com/dram/module/rdimm/m393a8g40ab2-cwe/.

[42] Haskell.org. *Haskell*. 2022. URL: https://www.haskell.org/.

[43] Technische Universität München – Fakultät für Informatik: Lehrstuhl III: Datenbanksysteme. *Umbra*. 2022. URL: https://umbra.db.in.tum.de.

[44] Technische Universität München – Fakultät für Informatik: Lehrstuhl III: Datenbanksysteme. *HyPer*. 2022. URL: https://www.hyper-db.de/.

[45] Oracle. *MySQL*. 2022. URL: https://www.mysql.com/.

[46] Oracle. *Oracle Database*. 2022. URL: https://www.oracle.com/database/.

[47] Microsoft. *SQL Server*. 2022. URL: https://www.microsoft.com/sql-server/.

[48] *SQL:2017 Standard. Database Languages–SQL–Part 6: SQL support for JavaScript Object Notation*. ISO/IEC TR 19075-6:2017.

[49] The PostgreSQL Global Development Group. *PostgreSQL version 11 — JSON Types*. 2018. URL: https://www.postgresql.org/docs/11/datatype-json.html.

[50] Oracle. *MySQL — JSON*. 2022. URL: https://dev.mysql.com/doc/refman/5.7/en/json.html.

[51] Oracle. *Oracle Database — JSON in Oracle Database*. 2022. URL: https://docs.oracle.com/en/database/oracle/oracle-database/21/adjsn/json-in-oracle-database.html.

[52] Microsoft. *SQL Server — JSON data in SQL Server*. 2022. URL: https://docs.microsoft.com/sql/relational-databases/json/json-data-sql-server.

[53] *SQL:1992 Standard. Database Languages–SQL*. ISO/IEC 9075:1992.

[54] The PostgreSQL Global Development Group. *PostgreSQL version 11 — **DECLARE***. 2018. URL: https://www.postgresql.org/docs/11/sql-declare.html.

[55] Oracle. *MySQL — Cursors*. 2022. URL: https://dev.mysql.com/doc/refman/5.7/en/cursors.html.

[56] Oracle. *Oracle Database — Explicit Cursor Declaration and Definition*. 2022. URL: https://docs.oracle.com/en/database/oracle/oracle-database/19/lnpls/explicit-cursor-declaration-and-definition.html.

[57] Surabhi Gupta, Sanket Purandare, and Karthik Ramachandra. "Aggify: Lifting the Curse of Cursor Loops Using Custom Aggregates". In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. SIGMOD '20. Portland, OR, USA: Association for Computing Machinery, 2020, pp. 559–573. ISBN: 9781450367356. DOI: 10.1145/3318464.3389736. URL: https://doi.org/10.1145/3318464.3389736.