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

Bachelorthesis Computer Science

Writing an Interpreter for a Database-Coupled Language

Xenia Wetzel

31.03.24

Examiner

Prof. Dr. Torsten Grust

Supervisor

Tim Fischer

Xenia Wetzel:
Writing an Interpreter for a Database-Coupled Language
Bachelorthesis Computer Science
Eberhard Karls Universität
From 29.11.23 to 31.03.24

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

Xenia Wetzel

Abstract

PL\Flummi is a research language for the compiler "Flummi". To verify the correctness of Flummi, we built an interpreter for PL\Flummi with the focal point of simplicity. With this in mind, we chose an abstract syntax tree (AST) interpreter, instead of the faster, but more complex, bytecode interpreter. Using a simple `match` statement, we traverse the AST of any PL\Flummi program. Now, comparing our interpreters output with Flummis output allows us to verify if Flummis output is correct. Rendering a few images using a ray tracer, we tested the performance of our interpreter, concluding that, except for very small inputs, our interpreter is significantly slower than Flummi. This emphasizes the fact that our interpreter is built with simplicity in mind, not performance. Comparing Flummis renders with those of our interpreter, we were able to verify Flummis renders as correct.

Contents

Abstract v

Acronyms ix

1 Introduction 1
- 1.1 Choosing an interpreter type 1

2 PL\Flummi 3
- 2.1 Grammar 3

3 Implementation 5
- 3.1 Evaluating values with DuckDB 5
  - 3.1.1 SQL expression to SQL statement 5
  - 3.1.2 Placeholders 6
- 3.2 From function call to Statement 6
- 3.3 Traversing the AST 8
  - 3.3.1 Assignment and Declaration 8
  - 3.3.2 Emit 8
  - 3.3.3 If 9
  - 3.3.4 Using exceptions for Stop, Loop, Continue and Break 9

4 Evaluation 11

5 Conclusion 13
- 5.1 Summary 13
- 5.2 Future Work 13

Bibliography 15

Acronyms

AST abstract syntax tree
CFG control flow graph

1 Introduction

Testing a compilers correctness is a crucial step in compiler development. In this bachelor thesis, we build a simple interpreter for the language PL\Flummi to test the correctness of the compiler “Flummi”.

Flummi is a compiler for iterative programs. It improves the performance of programs which need to access a database multiple times by generating an equivalent SQL query and therefore moving the whole program into the database. This reduces the number of times the program has to access the database and therefore increases performance. Flummi has a more involved implementation that can be split into two steps. It receives a control flow graph (CFG) as input, which it first transforms into a query. This query is then evaluated inside the database. This intermediate step leads to more opportunity for errors. The purpose of our interpreter is to produce a result in one simple step so it can be tracked easily, ensuring correctness. This way, the result of a program compiled by Flummi can be verified using our interpreter.

1.1 Choosing an interpreter type

There are different types of interpreters that can be used to interpret PL\Flummi. Considering the fact that our interpreter needs to be as simple as possible, we can compare the two main interpreter types there are: AST and bytecode interpreters. Both interpreter types require an AST as input.

Abstract syntax trees represent the structure of the program in form of a tree. They are called *abstract* syntax trees, because they leave out details of the program that are not integral to the structure, like indentation or comments [1, 2], as seen in the highlighted portions in Listing 1.1. For example, this simple program that adds 4 and 5 together.

```
1 # adding 4 and 5 together and emitting the result
2 CALL (§4§[], §5§[]) IN
3 FUN (a: §int§, b: §int§) -> §int§: '{
4     c: §int§;
5     c <- §{0} + {1}§[a, b];
6     EMIT §{0}§[c];
7     STOP
8 }
```

Listing 1.1: Simple function call in PL\Flummi with the arguments §4§[] and §5§[]. It adds the two given arguments together. Details that are not essential to the structure of the program are highlighted.

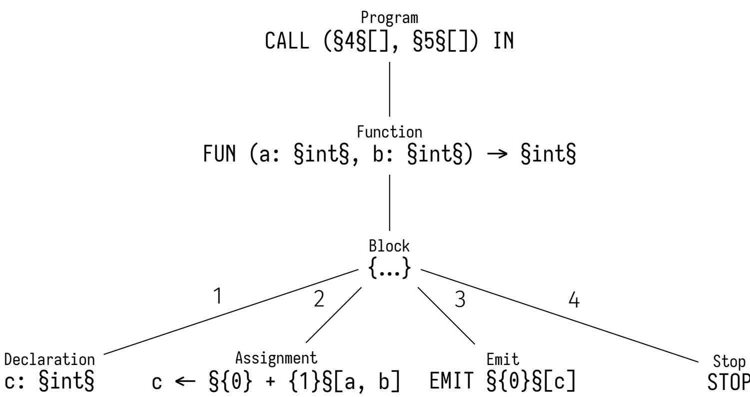

Figure 1.1: Abstract syntax tree of the function call in Listing 1.1.

As their name implies, an AST interpreter traverses the AST of a program to compute its results [1]. Bytecode interpreters also use ASTs, but they transform them into linear bytecode before executing the bytecode to produce results [3]. The advantage of bytecode interpreters over AST interpreters is that this linear byte code can be optimized more easily and therefore executing it is often significantly faster than going through it as written in the input language [1]. This advantage is not relevant for our interpreter because our goal is to keep the interpreter as simple as possible. The fact that bytecode interpreters first have to transform the AST to bytecode is just another source for errors and goes against the simple one-step solution that is needed. Iterating over an AST is more straightforward, which is why an AST interpreter is preferred here.

2 PL\Flummi

PL\Flummi is a research language developed for the compiler “Flummi”. It is a procedural language extension for SQL akin to extensions like PL/pgSQL, T-SQL and PL\SQL. Similar to those extensions [4, 5, 6], PL\Flummi also implements features like conditional statements and loops. Unlike all these extensions, PL\Flummi provides a lot more control. Each above-mentioned extension is coupled with its own DBMS. Flummi is supposed to be flexible in its use of DBMS, making it impossible to use an already established extension. On top of that, developing your own language means being able to change and extend the language as needed. In total, PL\Flummi provides all necessary features whilst keeping Flummi flexible.

2.1 Grammar

Program	P	:=	CALL (E, ..., E) IN F	
Function	F	:=	FUN ($v:\tau$, ..., $v:\tau$) -> $\tau:S$	
Statement	S	:=	{S; ...; S}	Block
		|	$v:\tau$	Declaration
		|	v <- E	Assignment
		|	IF v THEN S ELSE S	If
		|	LOOP v S	Loop
		|	CONTINUE v	Continue
		|	BREAK v	Break
		|	EMIT E	Emit
		|	STOP	Stop
Expression	E	:=	§<SQL expression>§[v, ..., v]	
Type	τ	:=	§<SQL type>§	
Variable	v	:=	<variable>	

Figure 2.1: Grammar of PL\Flummi.

Going over PL\Flummis grammar we can break down what the language is capable of. Every program P consists of one function call with a function declaration F inside. There cannot be more than one function call in a program. Since function call and function declaration are nested, there is also only one function declaration per program. The function body consists of one or more Statements S.

Key procedural language features include conditional statements and loops. PL\Flummi has If and Loop Statements. Inside the If Statement, it is important to note that the condition needs to be an already declared and assigned variable v instead of an Expression E. This has implications

for the implementation which we discuss later in Chapter 3. Looking at the Loop Statement including Break and Continue, a label is assigned to the loop. This way we can specify which loop to break or continue when working with nested loops.

Instead of return values, PL\Flummi has emit values, which are generated by the Emit Statement. These statements are similar to the `yield` keyword inside generators, where every `next()` generates a new value. In PL\Flummi the Emit Statement generates a new emit value each time the statement is passed. Unlike with `yield` however, the control flow is not paused after each Emit Statement, instead all emit values are generated with one function call. This can be implemented in different ways, which we discuss in Chapter 3. When we combine the Emit Statement with the Stop Statement, which ends the program execution, we can build something akin to a `return` statement.

Variables v in PL\Flummi have no scope. When statements start or end, declared variables stay inside the environment and can be accessed until we stop the program execution. They do have types, which means before each assignment we need to declare the variables type.

Expressions E are part of what makes PL\Flummi, and in turn Flummi, very flexible. The SQL expressions inside the paragraph signs are copied into the query that Flummi generates. We only need to make sure to write an expression that can be evaluated by our DBMS of choice. The rest of the query that Flummi generates uses SQL features from the SQL:1999 standard, which most modern DBMSs support.

3 Implementation

This chapter gives a more detailed explanation on how our interpreter is built. Since Flummi is implemented in Python, and therefore its parser and AST are as well, it is easiest to write our interpreter in the same language.

Before going over the implementation, we look at how Expressions are evaluated. To evaluate our Expressions, we need to decide on a DBMS. A popular choice is PostgreSQL. But PostgreSQL is a transactional DBMS [7], meaning it is more suited for many incoming queries instead of optimizing a handful of complex ones. We would also need to set up a server-client connection [8]. All this is not optimal for our purposes. DuckDB, on the contrary, is an analytical [9] DBMS that handles queries in-process [9] and in-memory [10], instead of on a server, making it very easy to use [10]. This makes DuckDB a far better solution for our interpreter.

Then, going over the implementation of our interpreter, we will first look at how we need to reconstruct the function call. Our interpreter can only traverse through Statements, and since the function call is not a Statement, we need to transform it. Finally, we look over how our interpreter traverses over the AST, going over each Statement.

3.1 Evaluating values with DuckDB

3.1.1 SQL expression to SQL statement

The SQL expressions contained within PL\Flummi cannot simply be given to DuckDB. Unlike SQL statements, SQL expressions on their own cannot be evaluated by a DBMS. To transform the SQL expression into a statement, we simply position a `SELECT` in front of it, changing it into a `SELECT` statement. Considering that `SELECT` statements can be nested, we also wrap brackets around the SQL expression, resulting in this final structure: `SELECT (<SQL expression>)`.

3.1.2 Placeholders

Because of implementation specifics inside Flummi, our SQL expressions contain Python string formatting placeholders instead of SQL variable placeholders. Before the SQL statements can be evaluated by DuckDB, the placeholders need to be changed to fit SQL standard. Placeholders in Python have their numbers wrapped in curly brackets. In SQL, a dollar sign precedes the number. Additionally, placeholder numbers in Python start with 0 whereas SQL standard starts with 1. To tackle these changes, each Python placeholder inside the Expression constructs an SQL placeholder in the shape of `($x :: type)`, making sure to start with placeholder number 1 instead of 0. Specifying the type of the placeholder is important, to ensure that our DBMS knows exactly how we wish to treat the value. These placeholders then replace the Python placeholders inside the SQL expression.

```
1  def expression_helper(self, expression: common.Expression[E]) -> str:
2      #transform free variables into correct SQL standard placeholders
3      placeholder_list = [
4          f"(${x + 1} :: {self.types[variable]})"
5          for x, variable in enumerate(expression.free_variables)
6      ]

7      #replace the current Python placeholders
8      temp_expression = expression.source.format(*placeholder_list)

9      #run SQL statement and return it
10     return duckdb.execute(
11         f"SELECT ({temp_expression})",
12         [self.env[x] for x in expression.free_variables]
13     ).fetchone()[0]
```

Listing 3.1: Implementation of how Expressions with placeholders are restructured to be evaluated by DuckDB.

For example, an SQL expression as seen in Listing 3.2 will be transformed into an SQL statement as shown in Listing 3.3.

```
1  {0} NOT BETWEEN 0 AND 1
```

Listing 3.2: Example SQL expression.

```
1  SELECT (($1 :: real) NOT BETWEEN 0 AND 1)
```

Listing 3.3: Transformed SQL expression from Listing 3.2 that can now be evaluated by DuckDB.

3.2 From function call to Statement

For us to be able to traverse the whole program with our interpreter, we need to bring the function call into a similar format as the rest of the program. The function call also includes the nested function definition. They are not Statements as seen in Section 2.1 and therefore cannot be traversed. Currently, a function call and definition include a number of arguments and

corresponding parameters. The easiest way to transform the function call into Statements is to wrap a Block Statement around the function body that includes declarations and assignments for each argument.

```
temp_block = proc.Block([])
inputs = program.inputs.copy()

for name, variable_type in f.parameters.items():
    temp_declaration = proc.Declaration(name, variable_type)
    temp_assignment = proc.Assignment(name, inputs.pop(0))
    temp_block.statements.append(temp_declaration)
    temp_block.statements.append(temp_assignment)

temp_block.statements.append(f.body)
```

Listing 3.4: Excerpt of the interpreter that transforms a function call to a series of `Statement` objects.

To show how exactly a program changes through this reconstruction, a program beforehand looks something like this.

```
CALL (§4§[], §5§[]) IN
FUN (a: §int§, b: §int§) -> §int§: {
    c: §int§;
    d: §int§;

    d <- §3§[];
    c <- §{0} + {1} + {2} + {1}§[a, b, d];

    EMIT §{0}§[c];
    STOP
}
```

Listing 3.5: An example function call that adds a few variables together including two that are given as arguments.

After being restructured, the function call and definition is gone and only Statements are left.

```
{
    a: §int§;
    a <- §4§[];
    b: §int§;
    b <- §5§[];

    {
        c: §int§;
        d: §int§;
        d <- §3§[];
        c <- §{0} + {1} + {2} + {1}§[a, b, d];

        EMIT §{0}§[c];
        STOP
    }
}
```

Listing 3.6: Transformed example from Listing 3.5 that can now be processed by the `match` statement.

3.3 Traversing the AST

With all the preparation out of the way, traversing the program can be done using a simple `match` statement over the AST, which now only holds Statement objects.

3.3.1 Assignment and Declaration

Both assignments and declarations are stored in global dictionaries. Because of PL\Flummis scoping, as discussed in Chapter 2.1, variables will remain inside these dictionaries until the program execution stops.

```
env: dict[common.Variable, any] = {}
types: dict[common.Variable, any] = {}
```

Listing 3.7: Implementation of dictionaries that hold variable values and types.

We need to declare the type of a variable, before the variable can be assigned. Assigning a variable is not as easy as assigning the Expression to the variable inside the dictionary. We need to evaluate the Expressions first, in case the program includes nested Expressions.

```
case proc.Declaration(variable, type):
    self.types[variable] = type.source
```

Listing 3.8: `match` case for Declarations.

```
case proc.Assignment(variable, expression):
    self.env[variable] = self.expression_helper(expression)
```

Listing 3.9: `match` case for Assignments.

3.3.2 Emit

Since the Emit Statement works similarly to the `yield` keyword in generators, as explained in Section 2.1, there are different ways to collect the various emit values. If we just print the values each time an Emit Statement is passed, we cannot return them at the end of the program execution. Instead, we collect the values inside a list which is returned and printed after the program has finished execution. By returning the list, it is possible to further work with the values outside the interpreter without having to change the implementation of the Emit Statement inside the interpreter itself.

```
return_list: list[any] = []
```

Listing 3.10: List that will be returned after a program is stopped, which holds all emitted values.

Similar to Assignments, we evaluate Emit Statements using DuckDB before storing them inside the `return_list`.

```
case proc.Emit(to_emit):
    result = self.expression_helper(to_emit)
    self.return_list.append(result)
```

Listing 3.11: `match` case for Emit.

3.3.3 If

Unfortunately, the If Statement has nothing new or exciting to demonstrate. Since the condition is a variable, which is evaluated on assignment, we simply access its value from `env`. With Pythons `if` statement, we then use the value to determine which branch to continue.

```
case proc.If(condition, t_branch, f_branch):
    if self.env[condition]:
        self.statement_helper(t_branch)
    else:
        self.statement_helper(f_branch)
```

Listing 3.12: `match` case for If Statements.

3.3.4 Using exceptions for Stop, Loop, Continue and Break

To implement the functionality of the Stop and Loop Statements, and by extension Continue and Break, we use exceptions to move up the call stack. Three different new exception types are declared to aid with this.

```
class LoopBreak(Exception): ...
class LoopContinue(Exception): ...
class Stop(Exception): ...
```

Listing 3.13: All newly defined exceptions inside the interpreter to aid with Loop and more specifically Break and Continue Statements as well as Stop Statements.

Starting with the implementation of Loop, Continue and Break. If we encounter a `CONTINUE` or a `BREAK` inside a Loop Statement, we raise the corresponding exceptions LoopContinue or LoopBreak. Together with the exception, we forward the label of the loop, which makes it possible to break or continue specific loops instead of just the loop in the current nesting level. The exception keeps getting raised through the possibly nested loops, and using Pythons `while` loop for the implementation, we can break or continue the labelled loop using Pythons `break` and `continue` keywords.

```python
case proc.Loop(name, body):
    while(True):
        try:
            self.statement_helper(body)
        except LoopBreak as e:
            if name == e.args[0]:
                break
            else:
                raise e
        except LoopContinue as e:
            if name == e.args[0]:
                continue
            else:
                raise e

case proc.Continue(name):
    raise LoopContinue(name)

case proc.Break(name):
    raise LoopBreak(name)
```

Listing 3.14: `match` case for Loop, Continue and Break Statements.

The implementation of Stop is much easier and follows the same principle. Inside the `match` statement we raise the Stop exception, which bubbles up the call stack and is then caught inside `interpret` causing the `return_list` to be returned. This marks the end of the program execution.

```python
case proc.Stop():
    raise Stop
```

Listing 3.15: `match` case for Stop.

```python
try:
    self.statement_helper(first_function_statement)
except Stop:
    return self.return_list
```

Listing 3.16: `try: ... except ...` statement, which initiates the `match` statement and returns the `return_list` after catching the Stop exception caused by the Stop Statement.

4 Evaluation

Writing a simple interpreter comes at the expense of performance. Looking back at Chapter 1, we choose an AST interpreter for its simplicity instead of the faster bytecode interpreter. In Chapter 3, we wrote a simple implementation to traverse the AST, making for a clear implementation but sacrificing performance. Now we want to see how bad the performance actually is and test if we can verify Flummis correctness.

Table 4.1: Image renders and performance using our interpreter and Flummi. Flummi (1-by-1) rendered the image by compiling a new query for each pixel. Flummi (batched) made a single query for the whole image. Since both outputs are the same, only the rendered image of Flummi (batched) is included. The difference column shows if there are differences between the renders of Flummi and our interpreter. Performance is measured in milliseconds. The number in brackets indicates how much slower the measured time is compared to the time of the interpreter.

Image Size	Renders		Difference	Performance		
	Interpreter	Flummi (batched)		Interpreter	Flummi (batched)	Flummi (1-by-1)
1×1				1 217ms	8 918ms (7.33×)	9 857ms (8.10×)
2×2				5 449ms	8 520ms (1.56×)	33 189ms (6.09×)
4×4				25 115ms	13 459ms (0.54×)	151 887ms (6.05×)
8×8				92 543ms	14 306ms (0.16×)	-
16×16				351 077ms	16 550ms (0.05×)	-
32×32				1 446 270ms	16 555ms (0.01×)	-

Using a ray tracer written in PL\Flummi, we render images, measuring the performance of our interpreter and Flummi. Flummi can render the images pixel by pixel, running a new query every time, or use its batch function to run only one query for all pixels. Surprisingly, our interpreter is better at small image sizes than Flummi, both 1-by-1 and even batched. Flummi (1-by-1) is significantly slower than both our interpreter and Flummi (batched), not improving even with bigger image size, which is why it is excluded from image size 8×8 and onwards. Flummi (batched) overtakes our interpreter at bigger image sizes, showing that Flummi scales well for larger program inputs, unlike our interpreter.

Looking at the renders and the difference image, we can determine if Flummis renders are correct. The difference image shows if the two renders differ. It is a ppm-file with two color-component values, 0 and 1. When the color-component value of a pixel in the two renders deviates, our respective difference image color component value is 1, otherwise it is 0. This leads to vibrant colors when there are even small differences in color-component values, making it easy to see.

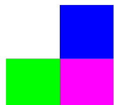

Figure 4.1: Difference image for two renders with deviating color-component values for each pixel.

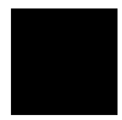

Figure 4.2: Difference image for two renders with the same color-component values for each pixel.

For our renders, no difference image shows any deviating values, meaning our interpreter and Flummi produced the same output. Therefore, Flummis renders are correct.

5 Conclusion

5.1 Summary

To summarize this work concisely, we built an AST interpreter for the language PL\Flummi to test the correctness of the compiler "Flummi". Our interpreters defining feature is being simple. PL\Flummis trait of being a research language allows for more control over its features, and therefore works well for a flexible compiler like Flummi. To evaluate Expressions, we use DuckDB for its simplicity and analytical approach to query optimization. The main implementation feature of the interpreter is the `match` statement, which traverses the AST of the program. To test our interpreter, we rendered images, concluding that Flummis output for those images is correct.

5.2 Future Work

Planned extensions for PL\Flummi include recursion and parallel computing. Currently, only one function call can be made per program. In the future we want to be able to make multiple function calls, including recursion. Parallel computing forks a task, so parallel processing is possible. Once these language features are implemented into PL\Flummi, we will extend this interpreter to continue verifying Flummis correctness.

Bibliography

[1] Octave Larose et al. "AST vs. Bytecode: Interpreters in the Age of Meta-Compilation". In: *Proc. ACM Program. Lang.* 7.OOPSLA2 (Oct. 2023), 233:3–233:4. DOI: 10.1145/3622808. URL: https://doi.org/10.1145/3622808.

[2] Joel Jones. "Abstract Syntax Tree Implementation Idioms". In: *Pattern Languages of Program Design* (2003). Proceedings of the 10th Conference on Pattern Languages of Programs (PLoP2003) http://hillside.net/plop/plop2003/papers.html, p. 2. URL: http://hillside.net/plop/plop2003/Papers/Jones-ImplementingASTs.pdf.

[3] Python Software Foundation. *Compiler design*. URL: https://devguide.python.org/internals/compiler/.

[4] The PostgreSQL Global Development Group. *PostgreSQL Documentation: PL/pgSQL — SQL Procedural Language*. URL: https://www.postgresql.org/docs/current/plpgsql.html.

[5] Orcale. *Oracle Database: Database PL/SQL Language Reference*. Mar. 2023. URL: https://docs.oracle.com/en/database/oracle/oracle-database/19/lnpls/index.html#Oracle%C2%AE-Database.

[6] Microsoft. *Microsoft SQL documentation: Transact-SQL reference (Database Engine)*. July 2023. URL: https://learn.microsoft.com/en-us/sql/t-sql/language-reference?view=sql-server-ver16.

[7] The PostgreSQL Global Development Group. *PostgreSQL Documentation*. URL: https://www.postgresql.org/docs/current/.

[8] The PostgreSQL Global Development Group. *PostgreSQL Documentation: Architectural Fundamentals*. URL: https://www.postgresql.org/docs/current/tutorial-arch.html.

[9] DuckDB Foundation. *DuckDB Documentation: Why DuckDB*. URL: https://duckdb.org/why_duckdb.

[10] DuckDB Foundation. *DuckDB Documentation: Python API*. URL: https://duckdb.org/docs/api/python/overview.