Compiling Datalog Programs by Interpreting Them
A Case Study on Abstraction without Regret

Yihong Zhang
yz489@cs.washington.edu

Abstract

In this paper, we examine the concept "abstraction without regret" and its applications in databases. We implement `Sdl` Datalog compiler in Scala, a programming language that supports many high-level abstractions, and achieves efficiency that is comparable to highly optimized C code. The key insight behind our approach is to automatically derive an efficient compiler from an interpreter, which is known as Futamura projection. We show our experience during the development of `Sdl`, to demonstrate the strengths of "abstraction without regret" as well as some current limitations. Finally, we benchmark on our tool and find that although preliminary results suggests that our tool beats state-of-the-art Datalog compilers and interpreters on many test cases, where it achieves up to 10X speedup. Although this does not necessarily imply that our tool is generally superior, our tool is promising and shows how efficiency can be achieved without losing high-level abstractions.

1 Introduction

It's known that most applications in today's database world is written in low-level languages, like C or C++. One reason for this is that most of today's databases can be dated back to decades ago, where new, high-level languages have not been invented and software engineering and organizations haven't been a big issue, while nowadays, modern database consists of millions lines of low-level code, which would be a disaster if not well-organized. As Christoph Koch put it,

> "In practice, the code of this core [of a classical SQL DBMS] is a great monolith, formidable in its bulk (millions of lines of code). Buffer and storage management are tightly integrated. The page abstraction is a prime example of abstraction failure, and stores identifiers relevant to other subsystems, such as recovery. Concurrency control rears its ugly head when it comes to protecting ACID semantics against insertions and deletions, and when there are multiple access paths to data, some hierarchical like B-trees. ..." [Koch (2013)]

Another reason, which is a more realistic consideration, involves performance issues. It's well known that performance is critical to database management system, and unfortunately, high-level programming languages, which come with its overhead, are notorious for its performance. This makes it impractical to write programs for efficiency-critical applications in a high-level language at the time.

However, as decades has grown, another research community, namely the PL community, has also gone a long way with countless brilliant ideas. Recent trends have proposed that now it's the time for both community to learn from each other for the development of modern, efficient database at a large scale with high productivity: equipped with the more developed PL techniques such as Futamura projection [Futamura (1999)], there has already been several works devoted to introducing indirections into database community without losing the efficiency competence, known as "abstraction without regret". Many previous works have been done in proposing such concept. For example, Amir Shaikhha *et al.* present in their award-winning VLDB '14 paper [(Shaikhha et al., 2018)] a query compiler written in Scala, a high-level, object-functional hybrid language

[Odersky and Rompf (2014)]. The compiler avoids the overhead of high-level languages while maintaining nicely-defined abstractions, beating cutting-edge query compiler written in low-level language. A replay of the same idea is also presented to the PL community as a Functional Pearl [Rompf and Amin (2015)], which implements a prototype SQL query compiler in less than 500 LOC, beating the highly optimized C code that makes up the bulk of MySQL and other traditional database systems.

In this paper, we're proposing, as a case study, a Datalog compiler in Scala called `Sdl`. Datalog is a Domain-Specific Language (DSL) for recursive query processing based on fixed-point semantics. Datalog has a very concise, yet expressive semantics, and has been applied widely to domains like program analysis, enterprise software, declarative networking, etc. Practical implementations of Datalog nowadays are usually Datalog-to-C compilers or interpreters, both implemented in a more traditional way. For example, Souffle [Jordan et al. (2016)] is an variant of Datalog targeting program analysis as well as general recursive query processing. It carries with itself an interpreter engine and a compiler engine in C++ (See Section 2).

Our main techniques in implementing the Datalog compiler is using Lightweight Modular Staging (LMS) [Rompf and Odersky (2012)]. LMS is a type-directed framework for staged compilation and generative programming. Empowered by LMS, we could synthesize efficient C programs from Datalog source code by specifying the semantics of Datalog with a standard definitional interpreter. Besides the efficient code synthesized, we obtain all the language features of the host language for free—subtyping, parametric polymorphism, higher order functions, to name a few—and all of these will be simply optimized away in the final C programs. This gives us the benefits of both the full abstractions of a high-level language and the efficiency comparable to these written in a low-level language.

2 Related Works

In this section, we outline some related works and a few backgrounds of the techniques we are presenting in this paper.

Partial Evaluations and Futamura Projection: Partial evaluation is the idea of partially evaluating the program before the runtime, most often during compilation time. This can be achieved because many evaluations are context independent and thus the result can be determined beforehand. For example, given the following program in Scala:

```
def f(x: Int) {
   def pow(n: Int, m: Int) =
      if (m == 0) 1 else {
         val temp = pow(n, m / 2)
         temp * temp *
            (if (isOdd(m)) n else 1)}
   val result0 = pow(3, 3)
   val result1 = pow(3, x)
   val result2 = pow(x, 3)
   // ...
}
```

It turns out that the value of `result0` can be calculated at compile time and the statement can be simplified as **`val`** `result = 28`. Furthermore, since we got partial clues for what values of `result1` and `result2` could be, we can indeed simplify the computation of them. The partially evaluated program will thus look like the following:

```
def f(x: Int) {
   def pow3(m: Int) =
      if (m == 0) 1 else
         val temp = pow(3, m / 2)
```

```
        temp * temp *
           (if (isOdd(m)) 3 else 1)}
   val result0 = 27
   val result1 = pow3(x)
   val result2 = (x * x) * x
   // ...
}
```

This could greatly benefit the runtime because the entire function no longer need to be evaluated from scratch every time the same program is run.

Now, what if we apply partial evaluations to interpreters? Suppose we have a partial evaluator called `Mix`, which accepts as first parameter the set of specializing parameters and the second parameter as the source program. It turns out that we obtain the ability of compiling any source code into target! More formally,

$$\text{Target} = [\![\texttt{Mix}(\text{Int}, \text{Src})]\!],$$

where Target is the target program, Int is the interpreter, and Src is the source program. This is called the first Futamura Projection, which has a history as old as the Relational Model [Futamura (1999). There are two other forms of Futamura Projection, derived by consistently applying `Mix` on the result obtained above, but for our use, it's enough to use this single form of Futamura Projection.

Lightweight Modular Staging (LMS): LMS is a framework for staged compilation. For arbitrary type `T`, LMS makes a distinction between `T`, which represents terms evaluated at generation time, and `Rep[T]`, which represents terms that will be generated. `Rep[T]` is automatically propagated through the type system, which guarantees the correct transitive closures of partial evaluations.

The empowering techniques behind LMS is Scala-Virtualized [Rompf et al. (2012)]. It combines shallow embedding and deep embedding, two DSL implementation options, by generalizing the concept of function and function calls to flow construct like `for` and `if`. This transforms every statement to be a pure function call, which could be further override to construct deep embedded Abstract Syntax Trees (AST). Therefore, it benefits from both the linguistic features of the host language and the opportunities for further optimizations and transformations. With Scala-Virtualized, LMS constructs deeply-embedded ASTs for the terms of `Rep[T]` type, which goes through multi-stage optimizations and finally transformed to target language source code by codegen.

Souffle: Souffle [Jordan et al. (2016)] is a variant of Datalog implemented in C++ that supports both running with interpreter mode and compiler mode. Given the source code, Souffle will first lower the declarative Datalog programs into execution plans in the form of an imperative intermediate representation (IR). After the IR is synthesized, Souffle will choose to either interpret the IR directly or compile them further down to C++ programs. The authors claimed that the key enabler for this transpilation is also Futamura projection. However, instead of utilizing the full strength of the compile-time computation, Souffle relies on the more limited C++ template metaprogramming for partially specifying semantics details. For example, Souffle uses the C++ template to specifying the schema of tuples that a B-tree should contain and the partial order on that tuple. However, the remaining parts, such as whole program structure, plan-execution logic, are specified with naive string concatenation.

String concatenation for source-to-source compilation has many problems: first, it may suffer from the scoping problems, because the host language scoping and the target language scoping are independent to each other and therefore many subtle issues may occur during the reasoning of the scoping: essentially, programmers are writing their programs in two orthogonal languages in a single file. Moreover, simple source-to-source compilation from the execution plan to target code, if done mechanically, loses further optimization opportunities at the target code level like dead code elimination or common subexpression elimination.

LegoBase, DBLAB, and LB2: These are both previous works in compiling SQL queries to low-level code using Scala. LegoBase [Shaikhha et al. (2018)] is the antecedent in proposing to use high-level language for compiling low-level code with Futamura projections. LegoBase and LB2 both use LMS as their backbones for Futamura projection and does the compilation in one pass, while DBLAB uses multiple passes and IRs to lower the abstraction level one stage by one stage, allowing for optimizations and refinements at different stages.

Similar to LegoBase and DBLAB but different from DBLAB, `Sdl` starts from the execution plan instead of the declarative program in Datalog. This is because there have been a large amount previous works done in synthesizing the optimal execution plan and the use of high-level language shouldn't be considered a barrier to be employed in these optimizers. Instead, the critical usage of low-level language is mainly from the concerns of efficiency, which is the problem many previous works on abstraction without regrets and we want to attack. Therefore, we choose to only focus on the phases after optimization has been applied. The difference between `Sdl` and LegoBase or DBLAB is that the execution plan we starts from is imperative and thus more low-level, having constructs like `for ... in`, `if` and their variants. The choice of imperative plan gives the optimizers more freedom to optimize the execution order while `Sdl` could be more concentrated at the code generation part.

3 Methodology

We use Souffle as our frontend parser of Datalog programs and synthesizer of execution plans. We parse the output of Souffle back to `Sdl` using Scala parser combinator library and then runs the parsed execution plan on a definitional interpreter to interpret the plan. The interpreter is abstract and only follows the minimal semantics the plan specifies and left many details for further interpretation. This gives programmers the freedom of substituting current implementations with new implementations and experiment with new implementations without tweaking the semantics. Then, we adapt the interpreter to the LMS framework and annotate only a few types with `Rep[T]` manually. Types of the remaining terms will be automatically inferred by Scala's type system.

LMS framework only provides basic Scala-to-C transformation building blocks so we need to extend LMS a little bit to provide constructs that can directly reflect corresponding C part. This is indeed very powerful because it provides the programmers with full control of all the low-level details, so that programmers loses nothing about implementation details but benefits from all the high-level abstractions (See section 4). Finally, all of the generation time abstractions like ones mentioned above will be optimized away by LMS.

As shown by the first Futamura projection, the adapted interpreter built upon LMS now can transform execution plans to efficient C programs. We thus obtained a compiler from the definitional interpreter for free.

4 Abstraction without regret

In this section, we discuss a few benefits of abstraction without regret that we learn from the implementation of `Sdl`, which includes zero-overhead abstraction and a full control of low-level details. We also name a few limitations of the LMS framework which is behind `Sdl` from a practical standpoint. These limitations are specific to the framework and implementation and won't generalize to other metaprogramming tools like C++ templates or MetaOcaml [Lilis and Savidis (2019)].

4.1 Zero-cost Abstraction

The emergence of system languages that provides language support for high-level abstractions demonstrates the popularity of zero-cost abstraction. For example, Rust provides facilities like algebraic datatype, higher-order functions and an advanced type system, while also ensures an execution speed that is competitive to C code. Our project provides an alternative approach to the same idea: instead of using a language that

Supported Statements and Operators		Description
Statement	`LOAD DATA FOR rel FROM filename`	load data
	`STORE DATA FOR rel TO filename`	store data
	`CLEAR rel`	clear a relation
	`SWAP (rel`$_A$`, rel`$_B$`)`	swap two relations
	`QUERY op`	embodies a query operator
	`LOOP stmts END LOOP`	infinite loop
	`EXIT cond`	exit the loop when `cond` holds
Operator	`FOR id IN rel`	relation scan
	`FOR id IN rel ON INDEX (id.i = expr`i`, ...)`	relation scan on index (i,...) with value (expr$_i$,...)
	`CHOICE id FROM rel WHERE cond`	find a value such that `cond` holds
	`CHOICE id FROM rel ON INDEX (id.i = expr`i`, ...) WHERE cond`	find a value on index (i,...) with value (expr$_i$,...) such that `cond` holds
	`IF cond`	proceeds if the condition holds
	`PROJECT (expr`i`,...) INTO rel`	Insert a tuple into a relation

Figure 1: Operators and Statements Supported by `Sdl`

has both features of a high-level and a low-level languages, we can separate our concerns for abstractions and efficiencies and address them in two different languages. Essentially, we programmed in one high-level source language and lower the program into the counterpart in a low-level target language. The soundness of the derived programs can be guaranteed as long as the original program and the translation process are sound.

In practice, this syntax-directed translation process is mechanical, because there are one-to-one mapping between the source and target language most of the time. When there isn't, it can be implemented. This provides a powerful mechanism for extending a low-level language with high-level abstractions. For example, consider the case that we want to use the higher-order function facility that is also dependent on the runtime in LMS (though we utilizes higher-order functions in `Sdl`, they do not occur in the generated C programs), we could implement closure conversions [Appel (1991)], which is an algorithm for compiling higher-order functions into first-order functions, as our translating function. This translation process could be seen as a parallel to macro expanding, which is also an approach to metaprogramming. However, LMS not only ensures execution efficiency of the generated code through partial evaluation, but also promises syntactic soundness like correct lexical scoping and semantic soundness, given the original program is sound (possibly guaranteed by the source language, e.g., type soundness) and the translation process is sound. What is also worth noticing is that LMS is implemented as a Scala library, which means that it's highly customizable in terms of the translation process and the target language options, making it as flexible as macro expanding.

Another benefits of our approach of using Futamura projection is that many abstractions can be optimized away. This property makes all the abstractions zero-cost as long as they are runtime independent. As it turns out in developing `Sdl`, most abstractions will exist only in the code generation time, because these abstractions are introduced for administrative sake, such as maintaining a manager interface of hash tables or information hiding of the underlying data structure implementations. Such abstractions are necessary because they more or less alleviate developers' burdens and increase productivity and maintainability. However, they will cause a substantial amount of runtime cost if not optimized away, even though they are not essential to the computation of final results.

4.2 Full control of low-level details

During the implementation of `Sdl`, we observe that it's very simple to extend the source language with target language features that does not has corresponding part in source language. For instance, a major difference between high-level languages like Scala and low-level languages like C is memory management, where high-level languages usually use automatic garbage collection and low-level languages require manual management. This problem can be handled easily by declaring a new node type with its associated constructor interface and code generation templates in LMS. This node type will be mixed into the constructed ASTs in LMS, which will be translated into target code per generation templates.

Taking advantage of the extensibility of LMS, we achieve a full control of low-level details, ignoring unwanted indirections. Consequently, the final generated programs can be precisely manipulated to achieve maximal flexibility. For instance, there are two ways of implementing arrays in C, i.e., by declaring a static array with fix size or `malloc`ing a new space and manually manage the memory. Static array allows for automatic resource memory management with scoping, while dynamical array allocation requires manual memory management, but is suitable for implementing growable lists. Such low-level manipulation decisions can be reflected with little effort in the implementation by substituting with appropriate code generation templates.

4.3 Limitations

We met some difficulties in implementing `Sdl`, which is mainly due to the indirections between the translation. For example, it's tricky to determine whether assigning a value to a variable denotes an assignment happening during code generation time (i.e., assigned during the execution of Scala metaprograms) or during the execution of the generated code (i.e., assigned during the execution of the compiled C programs). There are many other subtleties in the implementation, which are also hard to debug given the indirections between host and target languages.

5 Implementation details

Figure 2 shows the pipeline. The Datalog programs will be parsed into Souffle, which will go through multi passes for optimizations and be lowered into an execution plan. The execution plan IR outputted by Souffle encodes a semi-naïve evaluation strategy and consists of a set of common imperative operators like `FOR`, `CHOICE`, `IF` and their indexed version. Figure 1 shows statements and operators that are currently supported. Expressions in the table include constants, variable reference, common arithmetic operators and wildcard $\perp$, while conditions include boolean operators, inequality predicates and predicates for testing if a tuple is in a relation and if a relation is empty. Indexes will be automatically inferred from the entire execution plan.

The execution plan is then sent to `Sdl`, and `Sdl` will interpret the plan according to the semantics of statements and operators. Currently, `Sdl` uses a variant of naïve hash table for indexing and existential queries: for each relation, `Sdl` will create multiple indexing bucket arrays that all shares a same entry pool that contains tuples in the relation.

To implement the query compiler, we first implement an execution plan interpreter, in which high-level abstractions are widely used, including traits, inheritance, higher-order functions, and generics. These facilities hugely improves code reuse and conciseness. Then, to make our `Sdl` interpreter a compiler, We adapt it to the Lightweight Modular Staging (LMS) framework. We also add to LMS a moderate number of customized extensions that supports string hashing, string comparisons, etc. Given the interpreter and the Datalog program, LMS will apply many optimizations like common subexpression elimination and dead code elimination to synthesize an optimized C program, which will be compiled by a C compiler like `clang`.

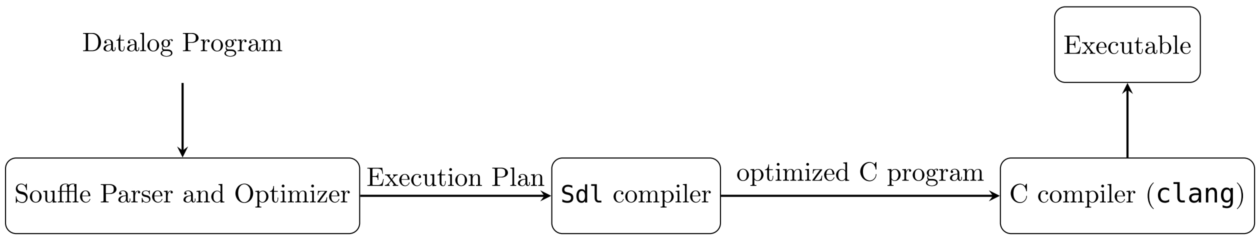

Figure 2: Overall pipeline of `Sdl`.

6 Results

All the benchmarks are tested on Macbook Pro mid-2015 version with Intel Core i7-4870HQ and 16GB memory.[1] We set the bucket size to be 2^{22} for default relations and 2^{14} for temporary relations (e.g. delta relations for each iteration). Also, we use

$$h(a_{1\dots n}) = p_1 * h(a_{1\dots n-1}) + a_n + p_2$$

as our hash function and pick two small prime 19260817 and 233 for p_1 and p_2.

The six test cases are collected from the original test suites that are used to evaluate Souffle [souffle lang ([n.d.])]. For each of the adapted benchmarks, we evaluate on the small-size (exp. $\leq$ 1min) case and medium-size cases (exp. $\leq$ 10 min) due to time limitations. Finally, we tested `Sdl` against executables generated by Souffle compiler and Datalog programs run under Souffle's interpreter mode.

Figure 3 shows the result. In particular, on all of the small cases, `Sdl` achieves 1.7X speedup compared to Souffle compiler and 14.5X speedup compared to Souffle interpreter. For medium-size cases, `Sdl` achieves speedups on all the test cases that it does not time out. Although lacking further investigations, we believe the time differences are rough and should largely be attributed to the parameters of hash tables and subtle performance gap between hash table and B tree, which is what the Souffle compiler and interpreter implement. As a side note, after increasing the bucket size for temporary relations, results for `Sdl` on topological_order (medium) and tc (medium) are improved from timeout to be 97.6s and 53.0s, achieving a speedup of 5.1X with regard to the Souffle compiler and 11.9X with regard to the Souffle interpreter.

It's also interesting to see that although the Souffle interpreter has a bad performance on most cases due to interpretation overhead, there is one case, namely cprog5 (medium) that Souffle interpreter performs better than other competitors, which timeout on this case. This unexpected result is reproduced for several times and we believe this may be due to runtime analysis and optimizations the interpreter does, which may not be implemented in the compiled version. It will thus be a future direction to work on add the support for runtime analysis of relation size and optimizes parameters to `Sdl` accordingly.

7 Discussion

It turns out that our evaluation is still preliminary and does not implies that our tool outperforms current state-of-the-art Datalog compilers and interpreters generally, although it does require radically less runtime on many test cases. We believe this is due to the instability of hash table, given that our hash table is implemented naively and does not support runtime optimizations. This is also suggested by the superiority of Datalog interpreter on cprog5 (medium) test cases. Therefore, a future direction will be to switch to a more robust hash tables and implement runtime optimizations for `Sdl`.

[1]We try to run the benchmark on `attu`, which is the only server machine we have access to, but some tests seem to never return on `attu`, possibly because `attu` will lower the priority of long-running tasks.

	Souffle Interpreter	Souffle Compiler	`Sdl` Compiler
cprog1 (small)	5.6s	1.0s	**0.2s**
cprog1 (medium)	267.1s	65.7s	**6.3s**
cprog5 (small)	106.1	6.3	**3.3s**
cprog5 (medium)	**332s**	TO	TO
tc (small)	15.0s	**3.3s**	**3.3s**
tc (medium)	951.5s	**195.5s**	TO
tak (small)	6.7s	0.6s	**0.2s**
tak (medium)	837.3s	62.2s	**27.8s**
topological_order (small)	10.4s	5.2s	**3.0s**
topological_order (medium)	836.5s	**579.2s**	TO
cellular_automata (small)	5.9	0.7	**0.3**
cellular_automata (medium)	345.4	38.5	**3.2**

Figure 3: Execution time for programs run by Souffle interpreter, C/C++ programs generated by Souffle compiler and `Sdl`. TO stands for timeout. In particular, on cprog5 (medium) and tc (medium), two cases where `Sdl` timeout, we achieve an execution time of 97.6s and 53.0s respectively after increasing the bucket size of hash tables for temporary relations.

8 Conclusion

We propose `Sdl` Datalog compiler, which achieves high efficiency that is comparable to state-of-the-art tools and uses high-level abstractions, which traditional database systems are likely to avoid. To implement `Sdl`, we simply implement a naiv̈e definitional interpreter. With the use of Futamura projection, this interpreter is transformed into a compiler that synthesizes highly-optimized C code from Datalog programs. This allows us to rely on many high-level abstractions in our implementation of the interpreter without affecting the efficiency of compiled code, since all these abstractions will be optimized away. Then, we discusses several benefits of our approach and benchmark on `Sdl`. It turns out that `Sdl` has a promising performance compared with state-of-the-art tools and we show some future directions that could effectively improve the efficiency and robustness of the `Sdl` Datalog compiler.

References

Andrew W. Appel. 1991. *Compiling with Continuations.* Cambridge University Press. https://doi.org/10.1017/CBO9780511609619

Yoshihiko Futamura. 1999. Partial Evaluation of Computation Process–An Approach to a Compiler-Compiler. *Higher-Order and Symbolic Computation* 12, 4 (1999), 381–391. https://doi.org/10.1023/A:1010095604496

Herbert Jordan, Bernhard Scholz, and Pavle Subotić. 2016. Soufflé: On Synthesis of Program Analyzers. In *Computer Aided Verification*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer International Publishing, Cham, 422–430.

Christoph Koch. 2013. Abstraction without regret in data management systems. In *CIDR*.

Yannis Lilis and Anthony Savidis. 2019. A Survey of Metaprogramming Languages. *ACM Comput. Surv.* 52, 6, Article Article 113 (Oct. 2019), 39 pages. https://doi.org/10.1145/3354584

Martin Odersky and Tiark Rompf. 2014. Unifying Functional and Object-Oriented Programming with Scala. *Commun. ACM* 57, 4 (April 2014), 76–86. https://doi.org/10.1145/2591013

Tiark Rompf and Nada Amin. 2015. Functional Pearl: A SQL to C Compiler in 500 Lines of Code. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP 2015)*. Association for Computing Machinery, New York, NY, USA, 2–9. https://doi.org/10.1145/2784731.2784760

Tiark Rompf, Nada Amin, Adriaan Moors, Philipp Haller, and Martin Odersky. 2012. Scala-Virtualized: Linguistic Reuse for Deep Embeddings. *Higher Order Symbol. Comput.* 25, 1 (March 2012), 165–207. https://doi.org/10.1007/s10990-013-9096-9

Tiark Rompf and Martin Odersky. 2012. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. *Commun. ACM* 55, 6 (2012), 121–130. https://doi.org/10.1145/2184319.2184345

Amir Shaikhha, Yannis Klonatos, and Christoph Koch. 2018. Building Efficient Query Engines in a High-Level Language. *ACM Trans. Database Syst.* 43, 1, Article Article 4 (April 2018), 45 pages. https://doi.org/10.1145/3183653

souffle lang. [n.d.]. Souffle Benchmarks. "https://github.com/souffle-lang/benchmarks".