Abstraction and Performance in Database Systems

Christoph Koch

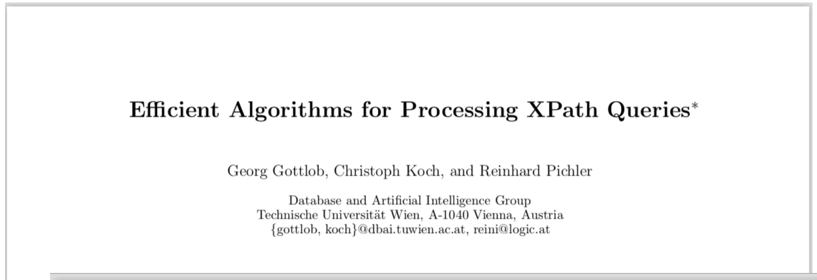
EPFL DATA Lab

Contents

My collaboration with Georg

20+ joint papers on expressiveness, complexity, and efficient evaluation of declarative/query languages.

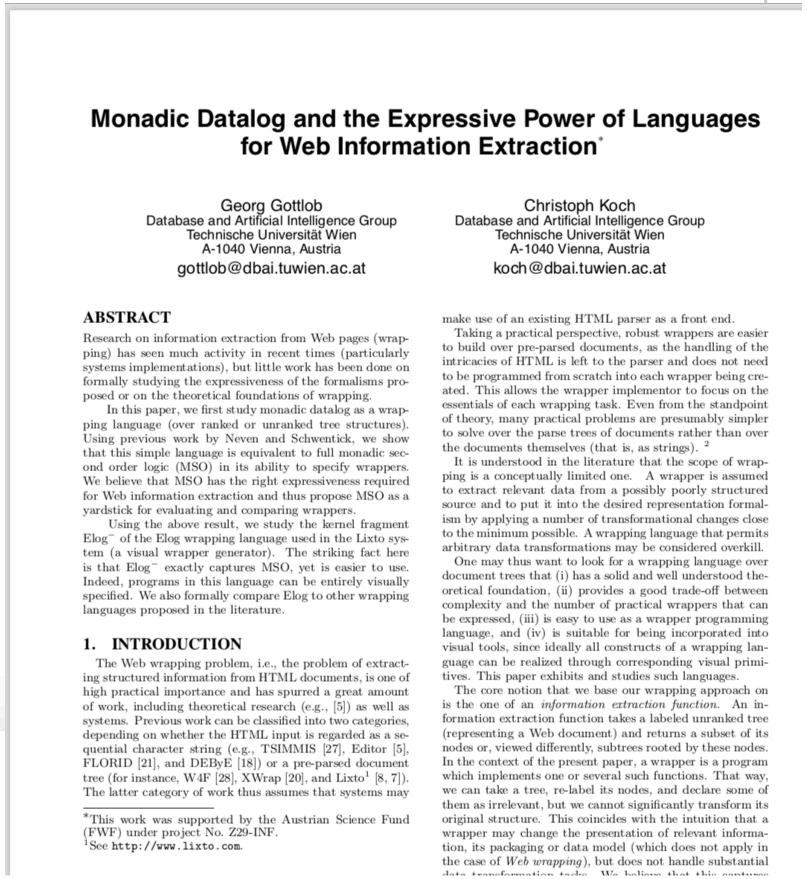
- Things I learned from Georg:
  - How to do research, really
  - How to write a PODS paper ☺
  - Using declarative languages creatively
  - Expressiveness vs. complexity is not a zero-sum game!
  - One can't just write papers and have a career here, but advance human knowledge!
  - Much more



Efficient Algorithms for Processing XPath Queries*

Georg Gottlob, Christoph Koch, and Reinhard Pichler

Database and Artificial Intelligence Group
Technische Universität Wien, A-1040 Vienna, Austria
{gottlob, koch}@dbai.tuwien.ac.at, reini@logic.at



Monadic Datalog and the Expressive Power of Languages for Web Information Extraction*

Georg Gottlob
Database and Artificial Intelligence Group
Technische Universität Wien
A-1040 Vienna, Austria
gottlob@dbai.tuwien.ac.at

Christoph Koch
Database and Artificial Intelligence Group
Technische Universität Wien
A-1040 Vienna, Austria
koch@dbai.tuwien.ac.at

ABSTRACT

Research on information extraction from Web pages (wrapping) has seen much activity in recent times (particularly systems implementations), but little work has been done on formally studying the expressiveness of the formalisms proposed or on the theoretical foundations of wrapping.

In this paper, we first study monadic datalog as a wrapping language (over ranked or unranked tree structures). Using previous work by Neven and Schwentick, we show that this simple language is equivalent to full monadic second order logic (MSO) in its ability to specify wrappers. We believe that MSO has the right expressiveness required for Web information extraction and thus propose MSO as a yardstick for evaluating and comparing wrappers.

Using the above result, we study the kernel fragment Elog$^-$ of the Elog wrapping language used in the Lixto system (a visual wrapper generator). The striking fact here is that Elog$^-$ exactly captures MSO, yet is easier to use. Indeed, programs in this language can be entirely visually specified. We also formally compare Elog to other wrapping languages proposed in the literature.

1. INTRODUCTION

The Web wrapping problem, i.e., the problem of extracting structured information from HTML documents, is one of high practical importance and has spurred a great amount of work, including theoretical research (e.g., [5]) as well as systems. Previous work can be classified into two categories, depending on whether the HTML input is regarded as a sequential character string (e.g., TSIMMIS [27], Editor [5], FLORID [21], and DEByE [18]) or a pre-parsed document tree (for instance, W4F [28], XWrap [20], and Lixto[1] [8, 7]). The latter category of work thus assumes that systems may make use of an existing HTML parser as a front end.

Taking a practical perspective, robust wrappers are easier to build over pre-parsed documents, as the handling of the intricacies of HTML is left to the parser and does not need to be programmed from scratch into each wrapper being created. This allows the wrapper implementor to focus on the essentials of each wrapping task. Even from the standpoint of theory, many practical problems are presumably simpler to solve over the parse trees of documents rather than over the documents themselves (that is, as strings). [2]

It is understood in the literature that the scope of wrapping is a conceptually limited one. A wrapper is assumed to extract relevant data from a possibly poorly structured source and to put it into the desired representation formalism by applying a number of transformational changes close to the minimum possible. A wrapping language that permits arbitrary data transformations may be considered overkill.

One may thus want to look for a wrapping language over document trees that (i) has a solid and well understood theoretical foundation, (ii) provides a good trade-off between complexity and the number of practical wrappers that can be expressed, (iii) is easy to use as a wrapper programming language, and (iv) is suitable for being incorporated into visual tools, since ideally all constructs of a wrapping language can be realized through corresponding visual primitives. This paper exhibits and studies such languages.

The core notion that we base our wrapping approach on is the one of an *information extraction function*. An information extraction function takes a labeled unranked tree (representing a Web document) and returns a subset of its nodes or, viewed differently, subtrees rooted by these nodes. In the context of the present paper, a wrapper is a program which implements one or several such functions. That way, we can take a tree, re-label its nodes, and declare some of them as irrelevant, but we cannot significantly transform its original structure. This coincides with the intuition that a wrapper may change the presentation of relevant information, its packaging or data model (which does not apply in the case of *Web wrapping*), but does not handle substantial

*This work was supported by the Austrian Science Fund (FWF) under project No. Z29-INF.
[1]See http://www.lixto.com.

The years 0 to 13AG

- I did more work on declarative languages
  - E.g. for probabilistic databases and video games
- I moved more into systems

- How could I combine declarative languages, expressiveness/efficiency with systems?
  - Domain-specific languages
  - Databases and compilation/code generation for performance.
- This is what I currently mostly do.

Declarative languages and DSLs

- Domain-specific languages (DSLs)
  - Engineered languages
  - Usually Turing-complete
  - Embedded DSL: classical PL (e.g. Java) + library (domain-specific vocabulary)

- SQL is a DSL (domain = database querying)
  - But most new DSLs are not very declarative.

- In Turing-complete DSLs: (compiler) optimizations tend to be local and sometimes brittle.

DSLs are hot!

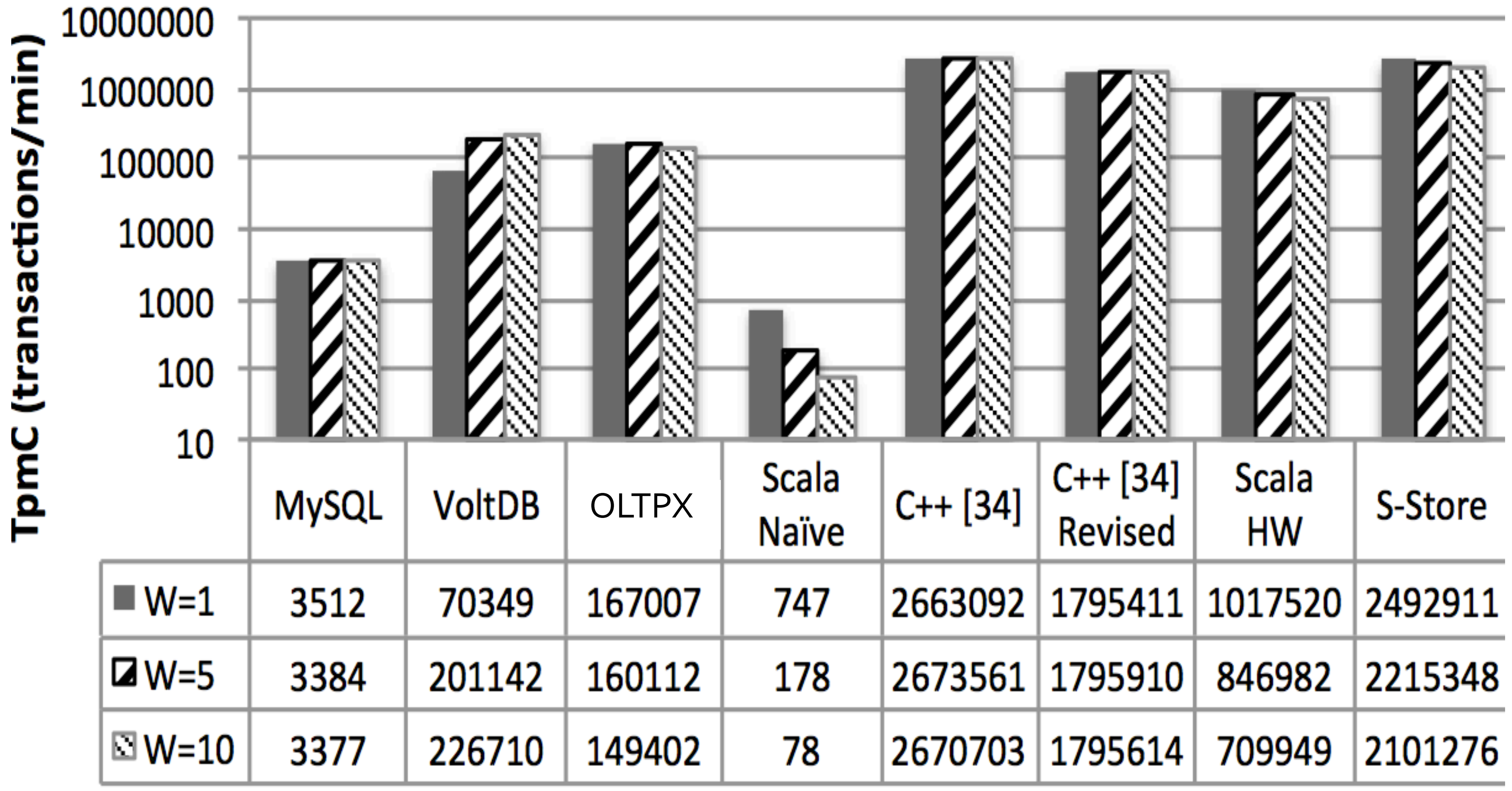
- Motivation: not declarativity but performance
  - Compensate for the failure of Dennard scaling and Moore’s law.
  - We don’t know how to build *robust* optimizing compilers with deep/global optimizations.
  - Consequence: Domain-specific compilation − opportunities for *automatic software specialization*.
- People all over CS are flocking to DSLs
  - Computer architecture. ASPLOS; Chisel, …
  - HPC & Graphics: OpenGL, Halide, …
  - Systems, databases: LegoBase, S-Store…

DSLs and code generation

- Software specialization by compilation.
  - Staging/partial evaluation (e.g. specialize DBMS code for a given schema).
- DSL compiler frameworks allow to easily add domain-specific code optimizations.
  - Usage in domain makes them robust.
  - Squid: github.com/epfldata/squid [Parreaux, Shaikhha, K., GPCE2017, Scala2017, POPL2018]

- Increasingly, DSLs enable code generation that matches or *outperforms* human systems programming experts!
  - Observed in multiple domains, e.g. linear transforms [Spiral], OLAP [LegoBase], OLTP [S-Store]
- “Abstraction without Regret” [Rompf&Odersky, CACM; K., CIDR2013]

S-Store TPC-C benchmark results



	MySQL	VoltDB	OLTPX	Scala Naïve	C++ [34]	C++ [34] Revised	Scala HW	S-Store
■ W=1	3512	70349	167007	747	2663092	1795411	1017520	2492911
◪ W=5	3384	201142	160112	178	2673561	1795910	846982	2215348
▨ W=10	3377	226710	149402	78	2670703	1795614	709949	2101276

Dashti, John, K., 2014

DSLs and the role of database research

- Relational databases created many firsts.
  - SQL is still the most successful DSL
  - RDBMS shows how to build an entire system, the entire stack, for executing SQL efficiently.
  - Algebras, plan languages, cost-based optimization, logical vs. physical data representation; managing the memory hierarchy, mem hierarchy-aware operator implementation.

– The basic pipeline and architecture is the foundation of all modern DSL-based systems.

– Some credit is given (e.g. GraphLab), **but** the database contribs are increasingly taken as a historical footnote across CS.

  - Also, are we still innovating in any significant way?

DSLs and the role of database research

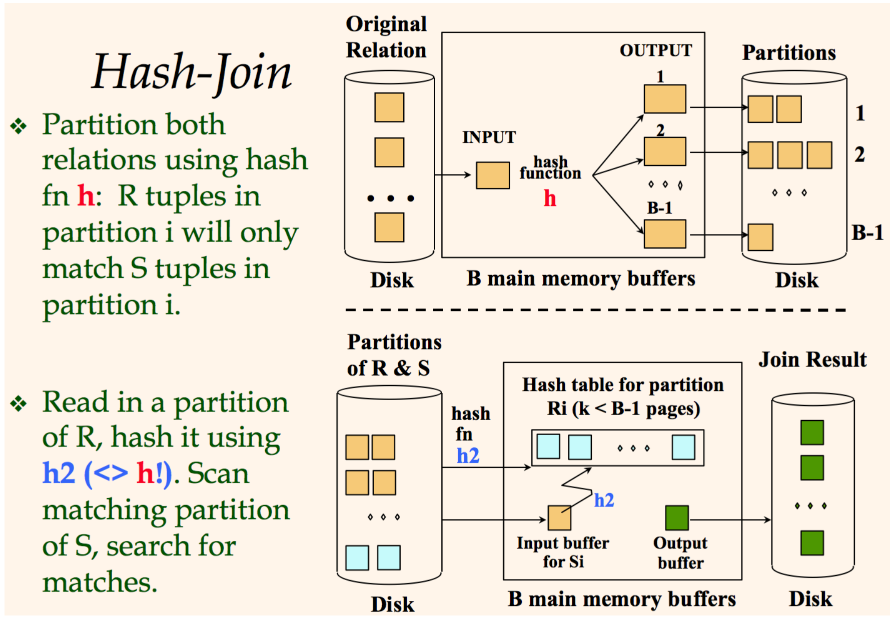
Database performance techniques are becoming mainstream … and the role of databases fades away. In two ways:

- The contributions of the DB community are becoming a historical footnote.
  - Database ideas stop being considered database ideas.
- Databases functionality is integrated into other kinds of systems, and classical DBMS will be used in fewer scenarios.

Example 1: row/columnar representations

- Much hyped (M. Stonebraker). Various DBMS built − Vertica, SAP Hana, …
- But: It's CS folklore now.
- Ubiquituous in programming tools
  - List<Pair<Int, Int>>: n+1 objects
  - Pair<List<Int>, List<Int>>: 3 objects
  - Makes a huge performance difference in OO runtime systems, e.g. JVM − boxing/unboxing overheads!!!
- Heavily used in HPC, graphics, ML, …

Example 2: GRACE Hash join



Hash-Join

- Partition both relations using hash fn **h**: R tuples in partition i will only match S tuples in partition i.
- Read in a partition of R, hash it using **h2 (<> h!)**. Scan matching partition of S, search for matches.

- Classical database course material. Seems uniquely about databases (?)
- Main-mem DB case: hash join becomes the trival implementation.
- GRACE hash-join = main mem hash join + staging for the mem hierarchy.
- Mem hierarchy considerations have by now been better analyzed/addressed by the compilers, computer architecture and HPC communities.
  - general/automatic algo transformation techniques exist (loop tiling & superoptimization; see Aho et al. Dragon Book 2nd Ed. Chapter 11)

A case of missing the boat

- Is there anything about DB Performance that won't be absorbed into the CS systems/performance mainstream?
- Conjecture: **No**.
- Experience in the DBLab project (github.com/epfldata/dblab) [Shaikhha, …, K., VDLB 2014, SIGMOD 2016, TODS2018, JFP2018].
  - We are building a library of compiler optimizations for data-intensive systems, by abstracting from a database system (LegoBase).
  - After cleaning up, none seem really specific to databases.

- This is a problem for the future of database research.

Database People Missing Boats Disorder (DMBD) – a pandemic?

- Causes:
  - Lack of care to recognize major CS trends (early)
  - Lack of effort to abstract&generalize results
  - Catering too much to reviewers in a calcified & broken system of conferences.

- Symptoms: Rectal pain, depression
- Treatment: ???

Another case of MtB in DB systems: NoSQL

- There always was distributed and parallel databases research.
  - Banned from first-rate publication venues
  - Few systems built − not “sexy” enough.
- Then Google and Facebook wanted scalable databases, and we couldn’t offer them.
- Consequences today:
  - A massive loss of prestige for our community
  - A widely-held belief that one has to look for SOSP rather than SIGMOD for good DB research.
  - Genuine contributions of the DB community do not get acknowledged and cited, but reinvented.

A third MtB case: DB Theory

- Estimated # of PODS papers talking of *DSLs*, ever: **0**
- Pub. venues for foundational DSL work: POPL, SIGGRAPH, ASPLOS, …
  - Citation in-degree into DB theory literature: ~0

Opportunities

- Many results from DB Theory, finite model theory on non-Turing complete languages carry over to modern DSLs.
- People in other domains do not know these results and find them exciting, when applied to their DSL.
- E.g. collection programming languages like Spark are essentially just nested relational algebra…
- My experience at a DSL summer school.

Quiz

From: *K, "Exploiting Domain-Specific Knowledge: […] Part 1: Lessons on DSLs learned by the DB community", DSL Design&Implementation Summer School, 2016.*

Consider the following DSL:

- purely functional Scala, with "if" as the only control structure
- Types built from Int, List, and tuples
- List ops: singleton constr, empty list, map(x => …) , flatten, list concat ++
- Tuple construction (…) and projection _i
- (deep) equality test =; the identity function

Let us call this language (Scala/List) **Monad Calculus (MC)** to have a label.

Example:

```
scala> val R = List(1)++List(2); val S = List(1)++List(3)
R: List[Int] = List(1, 2)
S: List[Int] = List(1, 3)

scala> R.map(r => S.map(s =>
             if (r==s) List((r,s)) else List()).flatten).flatten
res2: List[(Int, Int)] = List((1,1))
```

```
for(r <- R; s <- S; if (r==s)) yield (r,s)
Obviously, flatMap and filter are definable.
```

Quiz: What can you do in MC?

```
R.map(r => S.map(s =>
            if (r==s) List((r,s)) else List()).flatten).flatten
```

- Joins? --- **yes**
- Arbitrary "conjunctive queries" --- **yes**
- Arbitrary SQL select-from-where queries --- **no**, conditions (<)!
- Test whether two values are not equal --- **yes** (else)
- Test whether an item is *not* in a list -- **yes**(!) `List.filter(x=> x==a) == List()`
- Aggregations: select count(*) from … --- **no**
- Testing on order/look sideways, sorting a list of integers? --- **no**
- Reachability in a graph given by the edge relation? --- **no**

Quiz: What can you do in MC?

```
R.map(r => S.map(s =>
             if (r==s) List((r,s)) else List()).flatten).flatten
```

- Does every program terminate? --- **yes**
- How big is the largest value than can be produced? --- polynomial in input
- How quickly does every prog terminate? --- PTIME
- All queries of relational algebra --- **yes** !!!
- Only queries expressible in relational algebra --- **yes** % repr !!!!!!!!!!
- Can every program be parallelized? --- **yes**, fantastically well! (AC0)

-- given polynomially much hardware, every program runs in CONSTANT time!!!!

-- if you have only constantly much hardware => Brent Scheduling Principle

Quiz: Extending MC

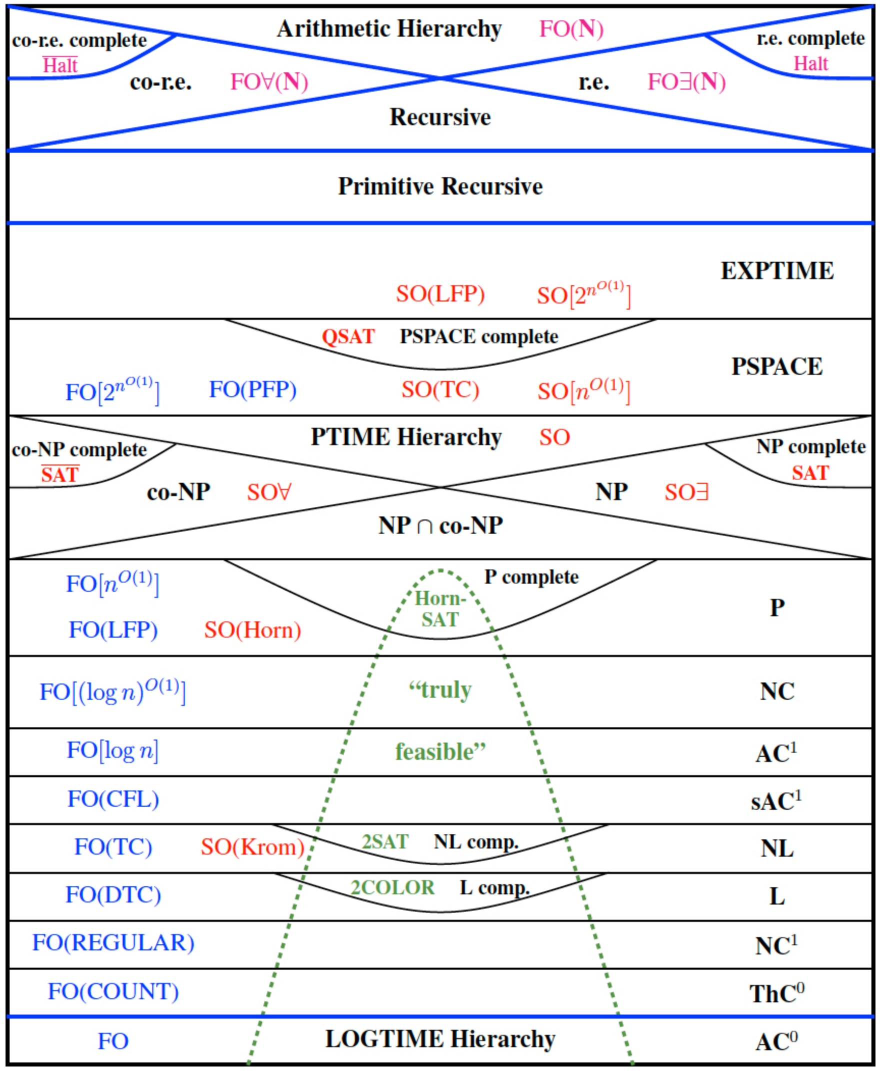
```
R.map(r => S.map(s =>
            if (r==s) List((r,s)) else List()).flatten).flatten
```

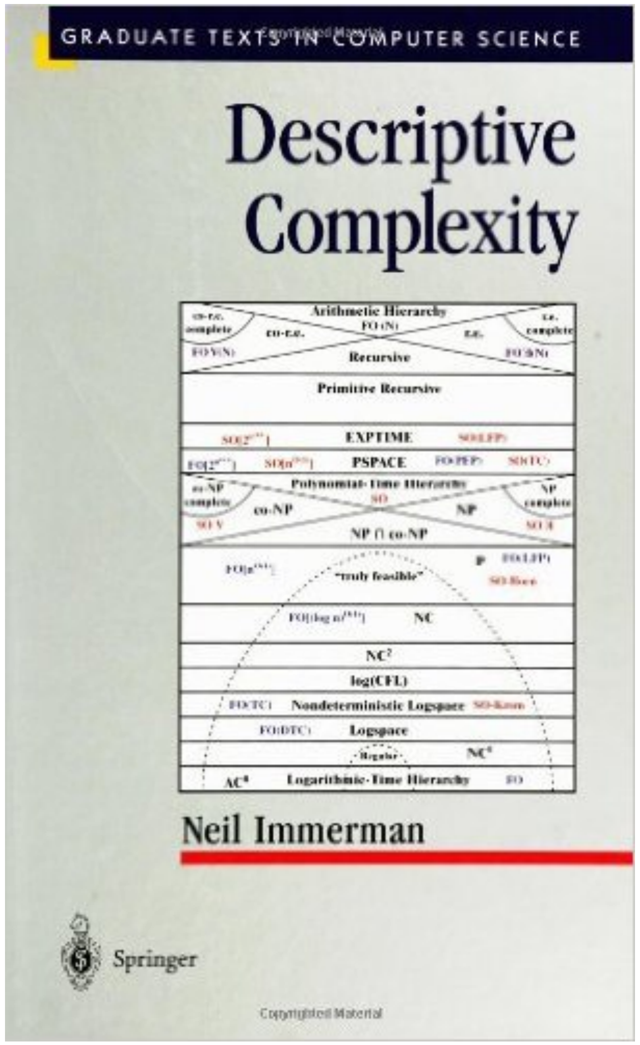
- Testing on order/look sideways, sorting a list of integers? --- **no**
- List.map preserves order but can't "query" it.
- But what if I want a DSL that can do this?

Could add List.foldLeft, and nothing else.

- Does every program still terminate? --- **yes**
- Does every program still run in PTIME? --- **no, nonelementary!**

The FO[X] DSL Zoo




GRADUATE TEXTS IN COMPUTER SCIENCE

Descriptive Complexity

Neil Immerman

Springer

Database theory work that we need more of

1. Results on complexity and *efficiency* that systems people can understand to be relevant to them, and which carry over to new languages, e.g.
   – Georg’s work on hypertree decompositions
   – Result cardinality bounds − AGM bound
   – Worst-case optimal joins
   – …
2. Results that bridge the gap between PODS and POPL/SIGGRAPH/ASPLOS work.

Summary

- Try not to miss the DSL boat.
- If this advice is useful to you, you ultimately have Georg to thank for it ☺