Inductive and Deductive Synthesis for Database Applications

by

John Killian Feser

B.S., Rice University (2015)
M.S., Rice University (2015)

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2023

Authored by:	John Killian Feser Department of Electrical Engineering and Computer Science May 18, 2023
Certified by:	Armando Solar-Lezama Professor of Electrical Engineering and Computer Science Thesis Supervisor
Accepted by:	Leslie A. Kolodziejski Professor of Electrical Engineering and Computer Science Chair, Department Committee on Graduate Students

Inductive and Deductive Synthesis for Database Applications

by

John Killian Feser

Submitted to the Department of Electrical Engineering and Computer Science
on May 18, 2023, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science and Engineering

Abstract

Program synthesis is a promising method for building efficient, flexible software by deriving low-level implementations from high-level specifications. In this thesis, I use programming-languages techniques to develop systems for synthesizing high-performance, specialized software and to build better general-purpose program-synthesis algorithms. I describe two new synthesis systems. First, I present a full-featured, synthesis-based pipeline for generating database implementations that are specialized to query workloads. This project shows that synthesis is a promising approach for building systems software, but building efficient synthesizers is still difficult, and in general a new synthesizer must be built for every new language. To address this need, I present a new, general-purpose inductive synthesizer, and show that it offers state-of-the-art performance on several challenging tasks.

Thesis Supervisor: Armando Solar-Lezama
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

First, I am deeply grateful to my advisor Dr. Armando Solar-Lezama for his support, advice, and relentless positivity. This work would not have been possible without his guidance.

I have had the pleasure of working with two wonderful collaborators: Dr. Sam Madden and Dr. Isil Dillig. Sam introduced me to the fascinating world of databases, and Isil's relentless work ethic made every paper I wrote with her stronger. I hope to continue to collaborate with both of you in the future.

Dr. Adam Chlipala read this thesis with a careful eye, and caught many mistakes. Any remaining errors are entirely mine.

The members of the Computer-Aided Programming group, both current and former, have been the source of much lively conversation and insight over the course of my Ph.D. I have made many friends in the programming-languages community at MIT—too many to list here.

Finally, my family is a constant source of encouragement and support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A key question in software systems research is: "Can we build systems (e.g. databases) that are both *fast* and *general-purpose*?" This thesis approaches this question from a programming-languages perspective by applying automated program-synthesis methods to the design of high-performance software systems. The promise of my research on program synthesis is to simplify systems design by helping programmers build specialized implementations that are exactly tailored to their specific use cases. Modern languages research has shown that compilers can be built for domain-specific languages (DSLs) that produce much better code than general-purpose compilers [57, 85]. Program synthesis offers a path towards building DSLs that allow programmers to write at a high level of abstraction; programs in synthesis-aided DSLs may be written declaratively instead of imperatively while still offering high performance [53], or they may leave some behavior unspecified so that it can be specialized later to the hardware or workload. The combination of synthesis and performance DSLs yields a powerful methodology for building high-performance software that can be rapidly developed and adapted to new use cases.

In this thesis, I use programming-languages techniques to develop systems for synthesizing high-performance, specialized software and to build better general-purpose program-synthesis algorithms. I developed a full-featured, synthesis-based pipeline (Chapter 2) for generating database implementations that are specialized to read-only query workloads [40]. This project shows that synthesis is a promising approach for building systems software, but building efficient synthesis systems is still difficult, and in general a new synthesizer must be built for every new language. This problem motivates my research into efficient, general purpose synthesis tools [37, 39, 41]. I developed a new, general-purpose inductive synthesizer (Chapter 3), and showed that it offers state-of-the-art performance on several challenging tasks.

1.1 Program Synthesis for Systems Applications

Program synthesis is a rapidly growing research field that has the potential to significantly lower the cost of software development [94, 45]. By deriving low level implementation details from high level specifications, program synthesis systems may someday automate the most tedious and time-consuming aspects of building software. The developer's responsibility shifts from implementation to specification, which has the following benefits:

- Synthesizers can offer strong guarantees that the resulting implementation correctly implements the specification. This can be achieved either by deriving implementations from specifications using semantics-preserving rules (deductive synthesis) or by using combinatorial search over program spaces (inductive synthesis).
- Specification development is often simpler and faster than developing an implementation. In some cases specifications can be extracted from existing software automatically, allowing programs to be revitalized by reimplementing their behavior for newer architectures [68].
- Software developed using synthesis can be quickly adapted to different hardware or workloads. Cost-guided synthesis approaches use cost models to derive efficient implementations [8, 59]. Changing the cost model and regenerating the implementation is a quick way to adapt software to new settings.

Program-synthesis algorithms can be classified into three broad categories: deductive, inductive, and hybrid.

Deductive synthesizers incrementally transform specifications into implementations by applying rules that decompose or transform the specification. Some deductive synthesizers are entirely user-guided; others use a combination of user guidance and search. When search is used, the goal is to select a sequence of transformations that translates the specification into a high-quality implementation.

Inductive synthesizers perform a search directly over some space of programs. This approach can make the synthesis problem more difficult, because every part of the solution must be generated from scratch. However, inductive synthesizers are straightforward to build, and they can be applied even when the specification is incomplete or may contain errors.

This thesis explores both inductive and deductive synthesis methods. In Chapter 4, I discuss ways that the two styles may complement each other, with a focus on database problems.

1.2 Deductive Program Synthesis of Relational Data Structures

Deductive approaches to program synthesis focus on deriving implementations from specifications by decomposing the specification into subproblems, solving the subproblems, and composing the results. Deductive synthesis has a long history [30, 12, 67] and has been successfully applied to problems in systems software construction ranging from transportation schedulers [6], to DSP transforms and FFTs [82], to embedded applications [23].

In the first part of this thesis, I show that deductive synthesis can be used to improve the performance of relational queries by selecting appropriate data structures. Relational queries (SQL) are a straightforward way to express computation in data-intensive applications. They have the pleasant property that the queries are largely independent of the way that the data is stored, as long as that storage can be accessed as if it were a relation. Database query planners offer a reasonably robust way to map between high-level queries and the low-level plans that operate on the particular data structures used. For example, database administrators can add an index to a relation and be confident that their queries will begin to use that index transparently, with the associated performance benefit. This *data independence* property makes it simple to rapidly develop applications that access and manipulate data. However, this flexibility is not without cost.

Databases are designed with highly dynamic query workloads in mind, which means that database designers choose general-purpose data structures that are useful for executing a wide range of queries but are rarely optimal for a particular query. Some applications truly need to execute arbitrary queries, but many applications have a fixed set of access patterns to their data. These patterns may change during development, but they are fixed once the application is deployed. We would like to offer developers flexibility during the exploratory phase of application development, while giving them the ability to trade that flexibility for performance during deployment.

In addition to flexibility during development, applications that are written at a high level of abstraction can be readily modified to adapt to changes.

In Chapter 2, I describe a new database system called Castor that is both flexible and offers significant improvements in read performance by specializing its internal data representation to a read-only workload. Castor uses deductive synthesis to generate a specialized data layout and query implementation from a parameterized query and database schema. The heart of the system is a new domain specific language called the *layout algebra*. The layout algebra can jointly express the layout of the data required to run a query along with the query operations themselves. This language was carefully designed to make it straightforward to write semantics-preserving rewrite rules that manipulate both the query and data layout simultaneously to express layout optimizations like the introduction of an index. Using this new language, I developed a cost-guided deductive synthesizer that transforms user specifications, written as SQL queries, into efficient query and layout plans.

The long-term goal of this project is to allow developers to write their applications at the same level of abstraction as database queries, while retaining the performance and space-efficiency of custom implementations. Database applications that are already written using queries are a natural fit for this technique, but the best candidates are programs that were not written using a database because the database would have been too costly. This style of synthesis has the potential to allow data-intensive systems applications to be written in a high-level style while achieving low-level performance.

Using synthesis to generate implementations of applications that are written using relational queries not a new idea, and Castor was inspired by a long line of work in deductive synthesis.

Fiat is one recent work in this space [28]. It is a deductive-synthesis tool embedded in the Coq proof assistant, and it offers tools for deriving implementations of abstract data types that are specified using a relational query language. In comparison with Castor, Fiat offers stronger correctness guarantees, because its generated code comes with a proof of correctness. Castor focuses on generating compact, efficient data structures and targets a higher level of automation than Fiat. It also supports a query language that is expressive enough to write standard database benchmark queries.

There have also been a number of attempts to apply inductive synthesis to this problem [65, 66, 111, 64]. While simpler to implement than deductive synthesizers, inductive synthesizers are difficult to scale, and their correctness guarantees rely on an automatic verifier. These inductive approaches either use a simplified query language or are unable to scale up to the queries in TPC-H, which is one of the simplest standard database benchmarks.

1.3 Efficient Inductive Program Synthesis with Distance Metrics

While inductive synthesis has proven difficult to scale up to handle entire SQL queries, it is still a powerful tool. In particular, inductive synthesis is a natural fit for problems where the specification is difficult to decompose deductively. In Chapter 3, I describe a new inductive-synthesis algorithm called SyMetric that outperforms existing approaches on three difficult problems: inverse constructive solid geometry, regular expression induction, and a task-planning problem. This algorithm is domain-agnostic, and it allows the user to give a distance metric that expresses the similarity between two programs. This metric is used to reduce the search space and to guide the search. Our approach was inspired by abstraction-guided synthesis, which uses abstractions to cluster programs that are similar in behavior [109]. In many domains, it is simpler to give a metric that expresses the similarity between programs than to give an abstraction.

While we have not applied SyMetric directly to problems in databases, in Chapter 4, I discuss how inductive-synthesis tools like SyMetric can be used to complement deductive-synthesis tools like Castor.

1.4 Summary of Contributions

I present two new program synthesis systems: Castor and SyMetric. I show that Castor can generate read-only database implementations that outperform high-quality traditional databases and that SyMetric is a general-purpose inductive synthesizer that outperforms the state of the art on two challenging synthesis tasks. In future work, I will continue to apply program synthesis to problems in database systems and to improve the state of the art in general-purpose program synthesis.

Chapter 2

Deductive Optimization of Relational Data Storage

2.1 Introduction

Traditional database systems are generic and powerful, but they are not well-optimized for static databases. A static database is one where the data changes slowly or not at all and the queries are fixed. These two constraints introduce opportunities for aggressive optimization and specialization. This paper introduces Castor: a domain-specific language and compiler for building static databases. Castor achieves high performance by combining query-compilation techniques from state-of-the-art in-memory databases [73] with a new deductive-synthesis approach for generating specialized data structures.

To better understand the scenarios that Castor supports, consider these two use cases. First, consider a company which maintains a web dashboard for displaying internal analytics from data that is aggregated nightly. The queries used to construct the dashboard cannot be precomputed directly, because they depend on parameters like dates or customer IDs, but all the queries are generated from a few query templates. Additionally, not all of the data in the original database is needed, and some attributes are only used in aggregates. As another example, consider a company which is shipping a GPS device that contains an embedded map. The map data is infrequently updated,

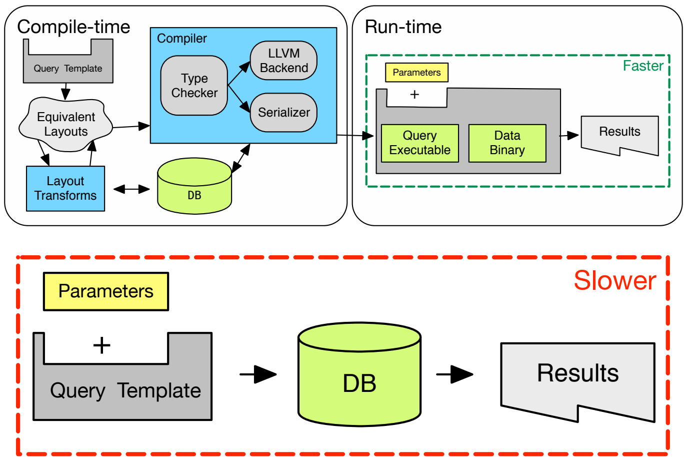


Figure 2-1: An overview of the CASTOR system (top) vs a traditional RDBMS (bottom).

and the device queries it in only a few specific ways. The GPS manufacturer cares more about compactness and efficiency than about generality. As with the company building the dashboard, it is desirable to produce a system that is optimal for the particular dataset to be stored.

These two companies could use a traditional database system, but using a system designed to support arbitrary queries will miss important optimization opportunities. Alternatively, they could implement their queries using custom data structures. This will give them tight control over their data layout and query implementation but will be difficult to develop and expensive to maintain.

CASTOR is an attempt to capture some of the optimization opportunities of static databases and to address the needs of these two scenarios. As Figure 2-1 illustrates, the input to CASTOR is a dataset and a parameterized query that a client will want to invoke on the data. The user then uses CASTOR's automatic optimizer or manually applies its high-level query transformations to generate an efficient implementation of an in-memory datastore specialized for the dataset and the parameterized query. The transformations available in CASTOR give the programmer tight control over the organization of the data in memory, allowing the user to trade off memory usage against query performance without the risk of introducing bugs. CASTOR uses code-generation techniques from high-performance in-memory databases to produce the low-level implementations required for efficient execution. The result is a package of data and code that uses significantly less memory than the most efficient in-memory databases and for many queries can surpass the performance of in-memory databases that already rely on aggressive code generation and optimization [73].

2.1.1 Contributions

Our primary contribution is the layout algebra: a new notation to jointly represent the layout of data in memory and the queries that will be computed on it. This joint representation allows us to write transformations that manipulate both the layout and the query in a single rewrite rule. This makes it easy to apply aggressive layout transformations.

We describe a set of deductive optimization rules for the layout algebra that generalize traditional query optimization rules to jointly optimize the query and the data layout and an automatic optimizer that applies these rules. We also implement a specializing layout compiler that produces both a binary representation of the data from the high-level data representation and machine code for accessing it.

Integrated Layout & Query Language We define the *layout algebra*, which extends the relational algebra [21] with layout operators that describe the particular data items to be stored and the layout of that data in memory. The layout algebra is flexible and can express many layouts, including row stores and clustered indexes. It supports nesting layouts, which gives control over data locality and supports prejoining of data. Our use of a language which combines query and layout operators makes it possible to write deductive transformations that change both the runtime query behavior and the data layout.

Automated Deductive Optimizer CASTOR provides a set of equivalence-preserving transformations which can change both the query and the data layout. The user can use CASTOR's optimizer to automatically select a sequence of transformations to deductively optimize their query. Alternatively, they can apply transformations manually to optimize without worrying about introducing bugs. Castor's notation turns transformations that would be complex and global in other database systems into local syntactic changes.

Type-driven Layout Compiler Existing relational-synthesis tools use standard-library data structures and make extensive use of pointer-based data structures that hurt locality [66, 65, 51]. Castor uses a specializing layout compiler that takes the properties of the data into account when serializing it. Before generating the layout, Castor generates an abstraction called a *layout shape* which guides the layout specialization. For example, if the layout is a row store with fixed-size tuples, the layout compiler will not emit a length field for the tuples. Instead, this length will be compiled directly into the query. This specialization process creates very compact datasets and avoids expensive branches in generated code.

High-performance Query Compiler Castor uses code-generation techniques from the high-performance in-memory database literature [73, 91, 101, 88]. It eschews the traditional iterator-based query-execution model [43] in favor of a code-generation technique that produces simple, easily optimized low-level code. Castor directly generates LLVM IR and augments the generated IR with information about the layout that allows LLVM to further optimize it.

Empirical Evaluation We empirically evaluate Castor on a benchmark derived from TPC-H, a standard database benchmark [25]. We show that Castor is competitive with the state of the art in-memory compiled database system Hyper [73] while using significantly less memory. We also show that Castor scales to larger queries than the leading data-structure synthesis tool Cozy [66].

2.1.2 Limitations

Castor *constructs read-only databases.* This design decision limits the appropriate use cases for Castor, but it enables important optimizations. Castor takes advantage of the absence of updates to tightly pack data together, which improves locality. Castor also aggressively specializes the compiled query by including information about the layout, such as lengths of arrays and offsets of layout structures. Providing this information to the compiler improves the generated code.

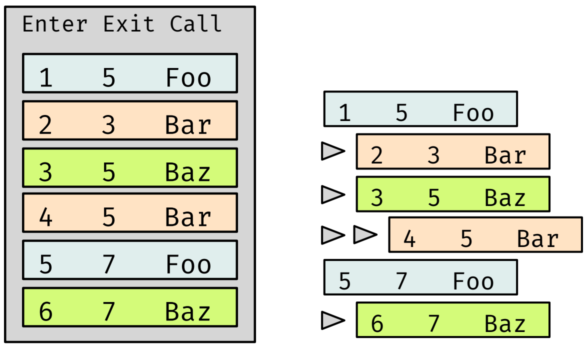
The optimizer processes one query at a time. Castor supports multiple-query workloads by reducing them to single-query workloads. However, the optimizer does not contain transformations that exploit possible sharing of layouts between different parts of a query, so the optimizer may replicate more data than necessary. However, Castor removes any data which is not needed by the query, and it produces compact layouts for the data that remains, which reduces the overhead of any replication.

2.2 Motivating Example

We now describe the operation of Castor on an application from the software-engineering literature. DemoMatch is a tool which helps users understand complex APIs using software demonstrations [112]. DemoMatch maintains a database of program traces—computed offline—which it queries to discover how to use an API. DemoMatch is a good fit for Castor because: (1) computing new traces is an infrequent task so the data in question is largely static and (2) the data is automatically queried by the tool, so there is no need to support ad-hoc queries. Finally, query performance is important for DemoMatch to work interactively.

2.2.1 Background

DemoMatch stores program traces as ordered collections of *events* (e.g., function calls). Traces have an inherent tree structure: each event has an `enter` and an `exit` and nested events may occur between the enter and exit. Figure 2-2 shows the table



(a) In tabular form. (b) In tree form.

Figure 2-2: Graphical representation of the DemoMatch data.

and tree structure of the DemoMatch data.

A critical query in the DemoMatch system finds nested calls to particular functions in a trace of program events:

$$\textsf{select}\ p.enter,\ c.enter\ \textsf{from}\ log\ \textsf{as}\ p,\ log\ \textsf{as}\ c\ \textsf{where}$$
$$p.enter < c.enter \land c.enter < p.exit \land p.id = \$pid \land c.id = \$cid$$

We refer to the caller as the *parent* function and the callee as the *child* function. Let p and c be the traces of events inside the parent and child function bodies respectively. The join predicate $p.enter < c.enter \land c.enter < p.exit$ selects calls to the child function from inside the parent function. The predicate $p.id = \$pid \land c.id = \$cid$ selects the pair of functions that we are interested in, where $\$pid$ and $\$cid$ are parameters.

2.2.2 The Layout Algebra

Castor programs are written in a language called the layout algebra. The layout algebra is similar to the relational algebra, but as we will see shortly, it can represent the layout of data as well as the operation of queries. By design, it is more procedural than SQL, which is more akin to the relational calculus [22]. For example, SQL leaves choices like join ordering to the query planner, whereas in the layout algebra join ordering is explicit.

In designing the layout algebra, we follow a well-worn path in deductive synthesis of creating a uniform representation that can capture all the refinement steps from a high-level program to a low-level one. Accordingly, the layout algebra can express programs which contain a mixture of high-level relational constructs and low-level layout constructs. At some point, a layout-algebra program contains enough implementation information that the compiler can process it. We say that these programs are *well-staged* (Section 2.3.4).

Here is the nested call query from Section 2.2.1 translated into the layout algebra:

$$\begin{array}{l}\texttt{select}(\{enter_p,\ enter_c\},\ \texttt{join}(enter_p < enter_c \wedge enter_c < exit_p, \\ \quad \texttt{filter}(\$pid = id_p,\ \texttt{select}(\{id \mapsto id_p,\ enter \mapsto enter_p,\ exit \mapsto exit_p\},\ log)), \\ \quad \texttt{filter}(\$cid = id_c,\ \texttt{select}(\{id \mapsto id_c,\ enter \mapsto enter_c\},\ log))))\end{array}$$

There are three layout-algebra operators in this query. $\texttt{filter}(p,\ r)$ filters the relation r by the predicate p. $\texttt{join}(p,\ r,\ r')$ takes the cross product of relations r and r' and filters the result by p. `select` takes a list of expressions with optional names and a query, and it selects the value of each expression for each tuple in the query, possibly renaming it.

The scoping rules for the layout algebra may look somewhat unusual, but they are intended to mimic the scoping conventions of SQL. In this query, the names *enter*, *exit* and *id* are field names in the *log* relation. The `select` operators introduce new names for these fields, using the $\mapsto$ operator, so that *log* can be joined with itself. We formalize the semantics of the layout algebra, including the scoping rules, in Section 2.3.2.

Note that at this point no layout is specified for *log*, so this program is not well-staged and so cannot be compiled. However, it still has well-defined semantics. In Sections 2.2.4 to 2.2.6, we describe how layouts can be incrementally introduced by transforming the program until it is well-staged.

2.2.3 Optimization Trade-Offs

The nested call query is interesting because the data in question is fairly large—hundreds of thousands of rows—and keeping it fully in memory, or even better in cache, is a significant performance win. Therefore, minimizing the size of the data in memory should improve performance.

However, there is a fundamental trade-off between a more compact data representation and allowing for efficient access. Sometimes the two goals are aligned, but often they are not. For example, creating a hash index allows efficient access using a key, but introduces overhead in the form of a mapping between hash keys and values.

In the rest of this section we examine three layouts at different points in this trade-off space: a compact nested layout with no index structures (Figure 2-3a), a layout based on a single hash index (Figure 2-3b), and a layout based on a hash index and an ordered index (Figure 2-3c). A priori, none of these layouts is clearly superior. The hash-based layout is the largest, but has the best lookup properties. The nested layout precomputes the join and uses nesting to reduce the result size, but it is more expensive for lookups. The last layout must compute the join at runtime, but it has indexes that will make that computation fast. The power of Castor is that it allows users to effectively explore different layout trade-offs by freeing them from the need to ensure the correctness of each candidate.

2.2.4 Nested Layout

Our first approach to optimizing the nested call query is to materialize the join, since joins are usually expensive, and to use nesting to reduce the size of the resulting layout.

The first step is to apply transformation rules (Section 2.4.2) to hoist and merge the filters. Now the join is in a term with no query parameters, so it can be evaluated at compile time:

$$
\begin{array}{l}
\texttt{select}(\{enter_p,\ enter_c\},\ \texttt{filter}(\$pid = id_p \land \$cid = id_c, \\
\quad \texttt{join}(enter_p < enter_c \land enter_c < exit_p, \\
\quad\quad \texttt{select}(\{id \mapsto id_p,\ enter \mapsto enter_p,\ exit \mapsto exit_p\},\ log), \\
\quad\quad \texttt{select}(\{id \mapsto id_c,\ enter \mapsto enter_c\},\ log))))
\end{array}
$$

After applying two more rules—projection to eliminate unnecessary fields (Section 2.4.3) and join elimination (Section 2.4.6)—the result is the following *layout program* (represented graphically in Figure 2-3a):

$$
\begin{array}{l}
\texttt{select}(\{enter_p,\ enter_c\},\ \texttt{filter}(id_c = \$cid \land id_p = \$pid, \\
\quad \texttt{list}(\texttt{select}(\{id \mapsto id_p,\ enter \mapsto enter_p,\ exit \mapsto exit_p\},\ log)\ \texttt{as}\ lp, \\
\quad\quad \texttt{tuple}_{\textsf{cross}}([\texttt{scalar}(lp.id_p),\ \texttt{scalar}(lp.enter_p), \\
\quad\quad\quad\quad \texttt{list}(\texttt{filter}(lp.enter_p < enter_c \land enter_c < lp.exit_p, \\
\quad\quad\quad\quad\quad \texttt{select}(\{id \mapsto id_c,\ enter \mapsto enter_c\},\ log))\ \texttt{as}\ lc, \\
\quad\quad\quad\quad \texttt{tuple}_{\textsf{cross}}([\texttt{scalar}(lc.id_c),\ \texttt{scalar}(lc.enter_c)])) \\
\quad\quad\quad\quad]) \\
\quad\quad)))
\end{array}
$$

In this program we see our first layout operators: $\texttt{list}(\cdot,\cdot)$ and $\texttt{tuple}_{\textsf{cross}}([\ldots])$.[1] The layout algebra extends the relational algebra with these operators, allowing us to write layout expressions, which describe how their arguments will be laid out in memory.

The above program can be read as follows. The operator $\texttt{list}(q\ \texttt{as}\ l,\ q')$ creates a list with one element for every tuple in q, and each element in the list is laid out according to q'. The outermost `list` in the program selects the *id*, *enter* and *exit* fields of the *log* relation and lays out each element of the list as a $\texttt{tuple}_{\textsf{cross}}$[2]. The

[1]As a point of notation, we separate layout operators from non-layout operators visually by **bolding** them. This is just to make the programs easier to read.

[2]In this expression, cross specifies how the tuple will eventually be read. Layout operators evaluate to sequences, so a tuple needs to specify how these sequences should be combined. In this case, we

first two elements in the tuple are the scalar representations of the id_p and $enter_p$ fields, and the third element is a nested list. Note that the content of that inner list is filtered based on the value of $lp.enter_p$ and $lp.exit_p$, and each element is laid out as a pair of two scalars id_c and $enter_c$.

The query is now well-staged because it satisfies the rules in Section 2.3.4. At a high-level, the rules require that we never use a relation without specifying its layout, a requirement that is satisfied in this case because all references to the log relation appear in the first arguments of `list` operators.

Figure 2-3a shows the structure of the resulting layout. This layout is quite compact. It is smaller than the fully materialized join because of the nesting; the caller `id` and `enter` fields are only stored once for each matching callee record.

To compare this query with the other queries that we consider, we benchmark it on a 92MB sample of DemoMatch data. [3] We find that it runs in 11.5ms and the generated layout takes up 50MB.

2.2.5 Hash-Index Layout

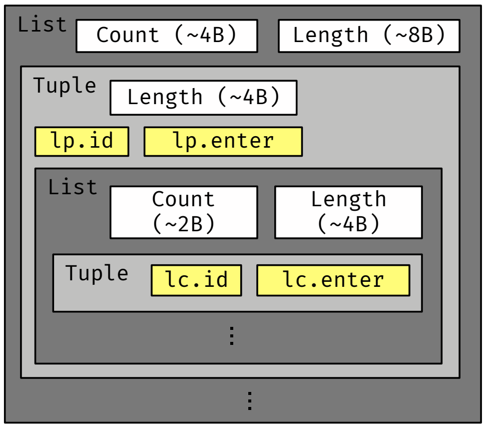
Now we optimize for lookup performance by fully materializing the join and creating a hash index. This layout will be larger than the nested layout, but lookups into the hash index will be quick, which will make evaluating the equality predicates on *id* fast. Figure 2-3b shows the structure of the resulting layout. When we evaluate the query, we find that it is much faster (0.4ms) but is larger than the nested query (60MB).

2.2.6 Hash- & Ordered-Index Layout

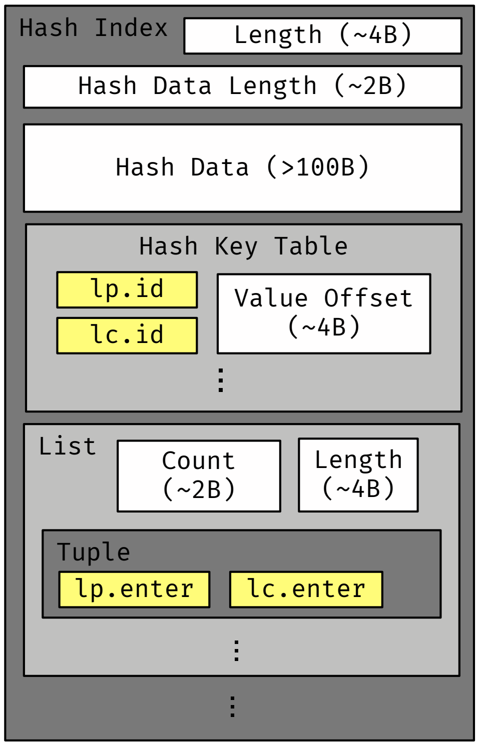
Finally, we investigate a layout (Figure 2-4) which avoids the full join materialization but still has enough indexing to be fast. We can see that the join condition is a range predicate, so we would like to use an index that supports efficient range queries to make that predicate efficient (Section 2.4.5). Then we can push the filters and introduce a hash table to select id_p. The resulting layout is shown in Figure 2-3c.

take a cross product.

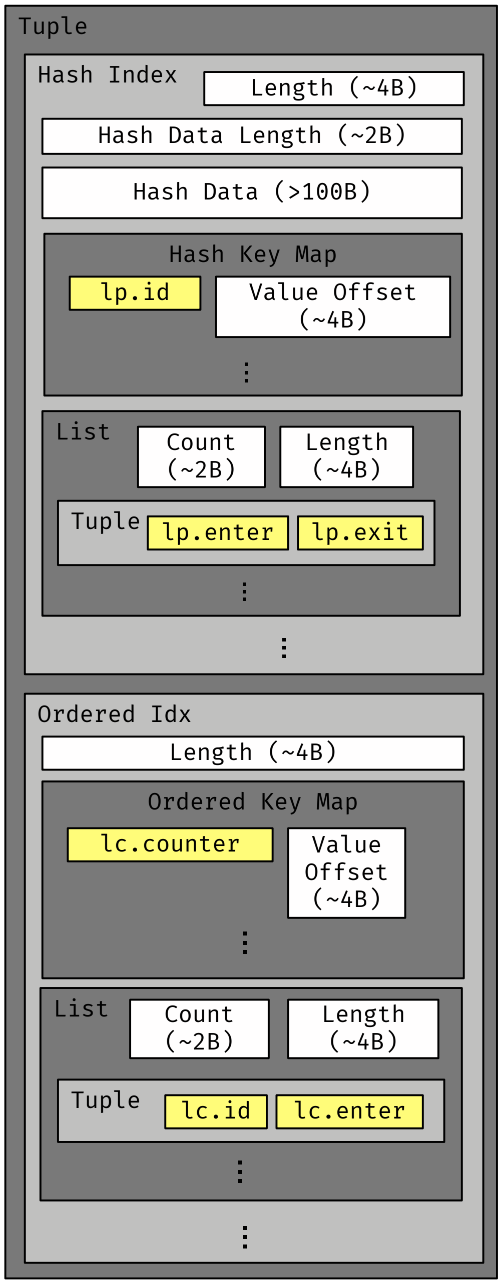
[3]All benchmarks are run on an Intel Xeon W-2155 with 64GB of memory.



(a) Nested.



(b) Hash-index.



(c) Ordered-index.

Figure 2-3: Layouts for the DemoMatch queries. The yellow boxes contain relational data, the white boxes contain metadata, and the gray boxes are the layout structure.

```
select({$enter_p$, $enter_c$},
  depjoin(hash-idx(select({$id$}, $log$) as $h$,
                   list(filter($h.id = id \wedge enter > exit$, $log$) as $lh$,
                        tuple$_{\textsf{cross}}$([scalar($lh.enter \mapsto enter_p$),
                                   scalar($lh.exit$)])),
                  $pid) as $p$,
          filter($id = \$cid$,
            ordered-idx(select({$enter$}, $log$) as $o$,
                        list(filter($enter = o.enter$, $log$) as $lo$,
                             tuple$_{\textsf{cross}}$([scalar($lo.id$),                ,
                                        scalar($lo.enter \mapsto enter_c$)]),
                        $p.enter_p$, $p.exit$))))
```

Figure 2-4: A layout that combines hash- and ordered-indexes.

This layout will be larger than the original relation but smaller than the other two layouts (9.8Mb), and it allows for much faster computation of the join and one of the filters (0.6ms).

This program introduces three new operators: `ordered-idx`, `hash-idx` and `depjoin`. `ordered-idx` creates indexes that support efficient range queries. It takes four parameters: *keys*, *values*, *upper*, and *lower*. *keys* is a relation that defines the set of keys and *values* is a dependent relation that defines the layout of values as a function of the keys. *lower* and *upper* are the bounds to use when reading the index ($p.enter_p$ and $p.exit$ in this case). `hash-idx`(*keys*, *values*, *lookup*) is similar, but it creates efficient point indexes using hash tables. *lookup* is the key to look up in the index (in this query, the key is $\$pid$).

The more interesting operator is the dependent join operator `depjoin`. In a dependent join, the right-hand-side of the join can refer to fields from the left-hand-side. In the `depjoin` operator, the left-hand-side is given a name (here it is p) that the right-hand-side can use to refer to its fields. One way to think about a dependent join is as a relational `for` loop: it evaluates the right-hand-side for each tuple in the left-hand-side, concatenating the results. Unlike the layout operators `list`, `hash-idx` and `ordered-idx`, `depjoin` executes entirely at runtime. It does not introduce any layout structure.

2.3 Language

In this section we describe the layout algebra in detail. The layout algebra starts with the relational algebra and extends it with layout operators. These layout operators have relational semantics, but they also have layout semantics which describes how to serialize them to data structures. The combination of relational and layout operators allows the layout algebra to express both a query and the data store that supports the execution of the query.

Programs in the layout algebra have three semantic interpretations:

1. The *relational semantics* describes the behavior of a layout-algebra program at a high level. We define this semantics using a theory of ordered finite relations [18]. According to this semantics, a layout-algebra program can be evaluated to a relation in a context containing relations and query parameters.

2. The *layout semantics* describes how the compiler creates a data file from a well-staged layout-algebra program. The layout semantics operates in a context which contains relations but *not* query parameters.

3. The *runtime semantics* describes how the compiled query executes, reading the layout file and using the query parameters to produce the query output. The runtime semantics operates in a context which contains query parameters but *not* relations.

These three semantics are connected: the layout semantics and the runtime semantics combine to implement the relational semantics. The relational semantics serves as

$$
\begin{aligned}
x &::= \text{identifier} \quad o ::= \textsf{asc} \mid \textsf{desc} \quad \tau ::= \textsf{cross} \mid \textsf{concat} \\
v &::= \text{integers} \mid \text{strings} \mid \text{Booleans} \mid \text{floats} \mid \text{dates} \mid \textsf{null} \\
e &::= v \mid x \mid x.x \mid e + e' \mid e - e' \mid e \times e' \mid e/e' \mid e \mathbin{\%} e' \\
&\mid \quad e < e' \mid e \leq e' \mid e > e' \mid e \geq e' \mid e = e' \\
&\mid \quad \texttt{if } e \texttt{ then } e_t \texttt{ else } e_f \\
&\mid \quad \texttt{exists}(q) \mid \texttt{first}(q) \mid \texttt{count}() \mid \texttt{sum}(e) \mid \texttt{min}(e) \mid \texttt{max}(e) \mid \texttt{avg}(e) \\
t &::= \{x_1 \mapsto e_1,\ \ldots,\ x_k \mapsto e_k\} \\
q &::= \emptyset \mid x \mid \texttt{dedup}(q) \mid \texttt{select}(t,\ q) \mid \texttt{filter}(e,\ q) \mid \texttt{join}(e,\ q,\ q') \\
&\mid \quad \texttt{group-by}(t,\ [x_1,\ \ldots,\ x_m],\ q) \mid \texttt{order-by}([e_1\ o_1,\ \ldots,\ e_m\ o_m],\ q) \\
&\mid \quad \texttt{depjoin}(q \texttt{ as } x,\ q') \mid \texttt{scalar}(x \mapsto e) \mid \texttt{tuple}_\tau(t) \mid \texttt{list}(q_r \texttt{ as } x,\ q) \\
&\mid \quad \texttt{hash-idx}(q_k \texttt{ as } x,\ q_v,\ e_k) \mid \texttt{ordered-idx}(q_k \texttt{ as } x,\ q_v,\ e_{lo},\ e_{hi})
\end{aligned}
$$

Figure 2-5: Syntax of the layout algebra.

a specification. An interpreter written according to the relational semantics should execute layout-algebra programs in the same way as our compiler.

2.3.1 Syntax

Figure 2-5 shows the syntax of the layout algebra. Note that the layout algebra can be divided into relational operators (`select`, `filter`, `join`, etc.) and layout operators (`list`, `hash-idx`, etc.). The layout algebra is a strict superset of the relational algebra. In fact, the layout operators have relational semantics in addition to byte-level data layout semantics (see Section 2.3.2).

2.3.2 Relational Semantics

The semantics operates on three kinds of values: scalars, tuples and relations. Scalars are values like integers, Booleans, and strings. Tuples are finite mappings from field names to scalar values. Relations are represented as lists of tuples. [ ] stands for the empty relation, : is the relation constructor, and $+\!\!+$ denotes the concatenation of relations.

We use sequences instead of sets for two reasons. First, sequences are more like bag semantics than the set semantics of the original relational algebra. This choice brings the layout algebra more in line with the semantics of SQL, which is convenient for our implementation. Second, sequences allow us to represent query outputs which have an ordering.

In the semantic rules, σ is a value environment; it maps names to scalar values. δ is a relation environment; it maps names to relations. We separate the two environments because the relation environment δ is global and immutable; it consists of a universe of relations that exist when the query is executed (or compiled) which are contained in some other database system. The value environment σ initially contains the query parameters, but some operators introduce new bindings in σ. $\cup$ denotes the binding of a tuple into a value environment. Read $\sigma \cup t$ as a new value environment that contains the fields in t in addition to the names already in σ.

In the rules, $\vdash$ separates environments and expressions, and $\Downarrow$ separates expressions and results. Read $\sigma,\ \delta \vdash l \Downarrow s$ as "the layout l evaluates to the relation s in the context $\sigma,\ \delta$."

We borrow the syntax of list comprehensions to describe the semantics of the layout-algebra operators. For example, consider the list comprehension in the `filter` rule (Equation (R-Filter)): $[t \mid t \leftarrow r_q \quad \sigma \cup t,\ \delta \vdash e \Downarrow \texttt{true}]$, which corresponds to the expression $\texttt{filter}(e,\ q)$. This list comprehension filters r_q by the predicate e where r_q is the relation produced by q. e is evaluated in a context $\sigma \cup t$ for each tuple t in r_q. Comprehensions that contain multiple $\leftarrow$, as in the `join` rule (Equation (R-Join)), should be read as the cross product that produces $[(x_1, y_1), (x_1, y_2), (x_2, y_1), (x_2, y_2)]$ from $[x_1, x_2]$ and $[y_1, y_2]$.

Finally, $\textsc{schema}(\cdot)$ is a function from a layout q to the set of field names in the output of q.

$$Id = (Scope?,\ Name) \quad Context = Tuple = \{Id \mapsto Value\} \quad Relation = [Tuple]$$

$$\sigma : Context \ \ \delta : Id \mapsto Relation \ \ s : Id \ \ t : Tuple \ \ v : Value \ \ r : Relation$$

$$\frac{t = \{n_1 \mapsto e_1, \ldots, n_m \mapsto e_m\} \qquad \forall i.\ \sigma,\ \delta \vdash e_i \Downarrow v_i}{\sigma,\ \delta \vdash t \Downarrow \{n_1 \mapsto v_1, \ldots, n_m \mapsto v_m\}} \quad \text{(E-Tuple)}$$

$$\frac{(x, r) \in \delta}{\sigma,\ \delta \vdash x \Downarrow r} \quad \text{(R-Relation)}$$

$$\frac{}{\sigma, \delta \vdash \emptyset \Downarrow [\,]} \quad \text{(R-Empty)}$$

$$\frac{\begin{array}{c} t \text{ contains no aggregates} \\ \sigma,\ \delta \vdash q \Downarrow r_q \qquad r = [t'' \mid t' \leftarrow r_q, \sigma \cup t', \delta \vdash t \Downarrow t''] \end{array}}{\sigma,\ \delta \vdash \texttt{select}(t,\ q) \Downarrow r} \quad \text{(R-Select)}$$

$$\frac{\begin{array}{c} \sigma,\ \delta \vdash q \Downarrow r_q \qquad\qquad \forall t \in r.\ t \in r_q \\ \forall t \in r_q.\ \exists i. 1 \leq i \leq |r| \wedge r[i] = t \wedge \forall j.\ j = i \vee t \neq r[j] \end{array}}{\sigma,\ \delta \vdash \texttt{dedup}(q) \Downarrow r} \quad \text{(R-Dedup)}$$

$$\frac{\sigma,\ \delta \vdash q \Downarrow r_q \qquad r = [t \mid t \leftarrow r_q \quad \sigma \cup t,\ \delta \vdash e \Downarrow \textsf{true}]}{\sigma,\ \delta \vdash \texttt{filter}(e,\ q) \Downarrow r} \quad \text{(R-Filter)}$$

$$\frac{\begin{array}{c} \sigma,\ \delta \vdash q \Downarrow r \qquad\qquad \sigma,\ \delta \vdash q' \Downarrow r' \\ s = [t \cup t' \mid t \leftarrow r, \sigma \cup t \cup t', \delta \vdash e \Downarrow \textsf{true}, t' \leftarrow r'] \end{array}}{\sigma,\ \delta \vdash \texttt{join}(e,\ q,\ q') \Downarrow r} \quad \text{(R-Join)}$$

$$\frac{\begin{array}{c} \sigma,\ \delta \vdash q \Downarrow r \qquad\qquad r' \text{ is a permutation of } r \\ r' \text{ is ordered according to the values of } e_1, \ldots, e_n \end{array}}{\sigma,\ \delta \vdash \texttt{order-by}([e_1 o_1, \ldots, e_n o_n],\ q) \Downarrow r} \quad \text{(R-OrderBy)}$$

Figure 2-6: Runtime semantics of the layout algebra.

$$\frac{\sigma,\ \delta \vdash q \Downarrow r \qquad r'' = \left[t' \;\middle|\; \begin{array}{l} t \leftarrow r, \\ t_s = \{s.f \mapsto v \mid (f \mapsto v) \in t\}, \\ \sigma \cup t_s,\ \delta \vdash q' \Downarrow r', \\ t' \leftarrow r' \end{array} \right]}{\sigma,\ \delta \vdash \texttt{depjoin}(q \texttt{ as } s,\ q') \Downarrow r''} \qquad \text{(R-Depjoin)}$$

$$agg(e, r) = \begin{cases} |r| & e = \texttt{count} \\ \min_{\sigma \cup t, \delta \vdash e' \Downarrow w, t \in r} w & e = \texttt{min}(e') \\ \max_{\sigma \cup t, \delta \vdash e' \Downarrow w, t \in r} w & e = \texttt{max}(e') \\ \sum_{\sigma \cup t, \delta \vdash e' \Downarrow w, t \in r} w & e = \texttt{sum}(e') \\ agg(\texttt{sum}(e'), r)/agg(\texttt{count}, r) & e = \texttt{avg}(e') \\ w \text{ s.t. } \sigma \cup t, \delta \vdash e \Downarrow w \text{ and } t \in r & \text{o.w.} \end{cases}$$

$$\frac{\sigma,\ \delta \vdash q \Downarrow r \qquad |r| = 0}{\sigma,\ \delta \vdash \texttt{group-by}(t,\ [\,],\ q) \Downarrow [\,]} \qquad \text{(R-GroupBy-1)}$$

$$\frac{\begin{array}{c} \sigma,\ \delta \vdash q \Downarrow r \qquad\qquad |r| > 0 \\ r' = [\{x_1 \mapsto agg(e_1, r), \ldots, x_m \mapsto agg(e_m, r)\}] \end{array}}{\sigma,\ \delta \vdash \texttt{group-by}(\{x_1 \mapsto e_1, \ldots, x_m \mapsto e_m\},\ [\,],\ q) \Downarrow r'} \qquad \text{(R-GroupBy-2)}$$

$$\frac{\begin{array}{c} q_k = \texttt{dedup}(\texttt{select}(\{y_1, \ldots, y_n\},\ q)) \\ q_v = \texttt{group-by}(E,\ [\,],\ \texttt{filter}(\bigwedge_{i=1}^{n} y_i = k.y_i,\ q)) \\ \sigma,\ \delta \vdash \texttt{depjoin}(q_k \texttt{ as } k,\ q_v) \Downarrow r \end{array}}{\sigma,\ \delta \vdash \texttt{group-by}(E,\ [y_1, \ldots, y_n],\ q) \Downarrow r} \qquad \text{(R-GroupBy-3)}$$

$$\frac{t \text{ contains aggregates} \qquad \sigma,\ \delta \vdash \texttt{group-by}(t,\ [\,],\ q) \Downarrow r}{\sigma,\ \delta \vdash \texttt{select}(t,\ q) \Downarrow r} \qquad \text{(R-SelectAgg)}$$

Figure 2-6: Continued runtime semantics of the layout algebra.

$$\frac{\sigma,\ \delta \vdash e \Downarrow v}{\sigma,\ \delta \vdash \texttt{scalar}(n \mapsto e) \Downarrow [\{n \mapsto v\}]} \quad \text{(R-Scalar)}$$

$$\frac{\sigma,\ \delta \vdash \texttt{depjoin}(q_r\ \texttt{as}\ s,\ q) \Downarrow r}{\sigma,\ \delta \vdash \texttt{list}(q_r\ \texttt{as}\ s,\ q) \Downarrow r} \quad \text{(R-List)}$$

$$\frac{}{\sigma,\ \delta \vdash \texttt{tuple}_\tau([\,]) \Downarrow [\,]} \quad \text{(R-Tuple-1)}$$

$$\frac{\sigma,\ \delta \vdash q_1 \Downarrow r_q \qquad \sigma,\ \delta \vdash \texttt{tuple}_{\mathsf{cross}}([q_2, \ldots, q_n]) \Downarrow r_{qs}}{\sigma,\ \delta \vdash \texttt{tuple}_{\mathsf{cross}}([q_1, \ldots, q_n]) \Downarrow [t \cup ts \mid t \leftarrow r_q \quad ts \leftarrow r_{qs}]} \quad \text{(R-Tuple-2)}$$

$$\frac{\sigma,\ \delta \vdash q_1 \Downarrow r_q \qquad \sigma,\ \delta \vdash \texttt{tuple}_{\mathsf{concat}}([q_2, \ldots, q_n]) \Downarrow r_{qs}}{\sigma,\ \delta \vdash \texttt{tuple}_{\mathsf{concat}}([q_1, \ldots, q_n]) \Downarrow r_q +\!\!+ r_{qs}} \quad \text{(R-Tuple-3)}$$

$$\frac{\begin{array}{c}\sigma,\ \delta \vdash \mathsf{depjoin}(q_k\ \texttt{as}\ s,\ \texttt{filter}(v_{lo} \le s.x \le v_{hi},\ q_v)) \Downarrow r \\ \sigma,\ \delta \vdash l_{lo} \Downarrow v_{lo} \qquad \sigma,\ \delta \vdash l_{hi} \Downarrow v_{hi} \qquad \text{SCHEMA}(q_k) = [x]\end{array}}{\sigma,\ \delta \vdash \texttt{ordered-idx}(q_k\ \texttt{as}\ s,\ q_v,\ l_{lo},\ l_{hi}) \Downarrow r} \quad \text{(R-OrderedIndex)}$$

$$\frac{\begin{array}{c}\sigma,\ \delta \vdash \texttt{depjoin}(q_k\ \texttt{as}\ s,\ \texttt{filter}(s.x = v,\ q_v)) \Downarrow r \\ \sigma,\ \delta \vdash l \Downarrow v \qquad\qquad \text{SCHEMA}(q_k) = [x]\end{array}}{\sigma,\ \delta \vdash \texttt{hash-idx}(q_k\ \texttt{as}\ s,\ q_v,\ l) \Downarrow r} \quad \text{(R-HashIdx)}$$

Figure 2-6: Continued runtime semantics of the layout algebra.

Relational Operators

First, we describe the semantics of the relational operators: `filter`, `join`, `select`, `group-by`, `order-by`, `dedup`, and `depjoin`. These operators are modeled after their equivalent SQL constructs.

`filter` filters a relation by a predicate e. `join` takes the cross product of two relations and filters it using a predicate e.

`select` is used for projection, aggregation, and renaming fields. It takes a tuple expression t and a relation r. If t contains no aggregation operators, then a new tuple will be constructed according to t for each tuple in r. If t contains an aggregation operator (`count`, `sum`, `min`, `max`, `avg`), then `select` will aggregate the rows in r. If t contains both aggregation and non-aggregation operators, then the non-aggregation operators will be evaluated on an arbitrary tuple in r.

`group-by` takes a list of expressions, a list of fields, and a relation. It groups the tuples in the relation by the values of the fields, then computes the aggregates in the expression list. `order-by` takes a list of expression-order pairs and a relation. It orders the tuples in the relation using the list of expressions to compute a key. `dedup` removes duplicate tuples.

Finally, `depjoin` denotes a dependent join, where the right-hand side of the join can depend on values from the left-hand side. It is similar to a for-each loop; `depjoin`(q `as` n, q')[4] can be read "evaluate q' for each tuple in q and concatenate the results." We use `depjoin` as a building block to define the semantics of the layout operators.

Layout Operators

The novelty of the layout algebra is that it can express the layout of data in addition to queries over that data. We introduce the following operators for describing data layouts: `scalar`, `tuple`, `list`, `hash-idx`, and `ordered-idx`. We chose these layout primitives because they are compositional, have good spatial locality and support

[4]In this expression, n is a *scope*, and it qualifies the names in q.

common query patterns such as range and equality predicates.

The layout operators have relational semantics. Although the layout operators can be used to construct complex, nested layouts, they evaluate to flat sequences of tuples of scalars, just like the relational operators. The rules in this section only describe the relational behavior of the layout operators; they do not address the question of how data is laid out or how it is accessed. We discuss these aspects of the layout operators in Section 2.3.3.

The simplest layout operators are `scalar` and `tuple`. Evaluating a `scalar` operator produces a relation containing a single tuple. The `tuple` operator represents a fixed-size, heterogeneous list of layouts. When evaluated, each layout in the tuple produces a relation, which are combined either with a cross product or by concatenation.

Note that evaluating a `tuple` operator produces a *relation* not a tuple. Although these semantics are slightly surprising, there are two reasons why we chose this behavior. First, it is consistent with the other layout operators, all of which evaluate to relations. Second, `tuple`s can contain other layouts (`list`s for example) which themselves evaluate to relations.

Standard (non-nested) database tuples are represented by tuples of scalars, combined using a cross product. Each scalar evaluates to a singleton relation, so the cross product produces a relation containing a single tuple.

The remaining layout operators—`list`, `hash-idx` and `ordered-idx`—have a similar structure. We discuss the `list` operator in detail. `list` is essentially an alias for `depjoin`. Like `depjoin`, `list` takes two arguments: q_r and q. These two arguments should be interpreted as follows: q_r describes the data in the list. Each element of the list has a corresponding tuple in q_r, so the length of the list is the same as the length of q_r. One can think of each tuple in q_r as a kind of key that determines the contents of each list element. On the other hand, q describes how each list element is laid out. q will be evaluated separately for each tuple in q_r. It determines for each "key" in q_r what the physical layout of each list element will be, as well as how that element must be read.

Returning to the query in Section 2.2.4, the inner `list` operator

$$\begin{aligned}&\texttt{list}(\texttt{filter}(lp.enter_p < enter_c \land enter_c < lp.exit_p,\\ &\qquad\qquad \texttt{select}(\{id \mapsto id_c,\ enter \mapsto enter_c\},\ log))\ \texttt{as}\ lc,\\ &\quad \texttt{tuple}_{\mathsf{cross}}[\texttt{scalar}(lc.id_c),\ \texttt{scalar}(lc.enter_c)])\end{aligned}$$

selects the tuples in *log* where $enter_c$ is between $enter_p$ and $exit_p$ and creates a list of these tuples. The first argument describes the contents of the list and the second describes their layout. This program will generate a layout that has a list of tuples, structured as $[(id_{c1},\ enter_{c1}), \ldots, (id_{cn},\ enter_{cn})]$.

`hash-idx` and `ordered-idx` are similar to `list`. They have a query q_k that describes which keys are in the index and a query q_v that describes the contents and layout of the values in the index.

For example, in:

$$\begin{aligned}&\texttt{hash-idx}(\texttt{select}(\{id\},\ log)\ \texttt{as}\ h,\\ &\qquad \texttt{list}(\texttt{filter}(\{id = h.id\},\ log)\ \texttt{as}\ l,\\ &\qquad\qquad \texttt{tuple}_{\mathsf{cross}}[\texttt{scalar}(l.enter),\ \texttt{scalar}(l.exit)]),\ \$pid)\end{aligned}$$

the keys to the hash-index are the *id* fields from the *log* relation. For each of these fields, the index contains a list of corresponding (*enter*, *exit*) pairs, stored in a tuple. When the hash-index is accessed, $\$pid$ is used as the key. This program generates a layout of the form: $\{id \mapsto [(enter,\ exit), \ldots], \ldots\}$, which is a hash-index with scalars for keys and lists of tuples for values.

Scopes & Name Binding

The scoping rules of the layout algebra are somewhat more complex than the relational algebra. There are two ways to bind a name in the layout algebra: by creating a relation or by using an operator which creates a scope.

All the operators in the layout algebra return relations. Some operators simply pass through the names in their parameter relations. Others, such as `select` and `scalar`, can be used for renaming or for creating new fields.

Some operators, such as `depjoin`, create *scopes*. A scope is a tag which uniquely identifies the binding site of a name. For example, in `depjoin`(q `as` s, q'), a field f from q is bound in q' as $s.f$. Scoped names with distinct scopes are distinct, and scoped names are distinct from unscoped names. We add scopes to the layout algebra as a syntactically lightweight mechanism for renaming an entire relation. Renaming entire relations is necessary because shadowing is prohibited in the layout algebra. Prohibiting shadowing removes a major source of complexity when writing transformations. While we could use `select` for renaming, we opted to add scopes so that renaming at binding sites would be part of the language rather than a pervasive and verbose pattern.

There are still situations when renaming entire relations using `select` is necessary. For example, in a self-join one side of the join must be renamed.

2.3.3 Preview of Layout & Runtime Semantics

In this section, we give a preview of the layout and runtime semantics, which are discussed in detail in Section 2.6.1 and Section 2.6.3. The layout semantics specifies the layout of data in memory at a byte level. Each layout operator has a serialization format, and the semantics describes how these formats are composed together. The runtime semantics describes how the layout and query operators read the data from that serialization format and produce a query output.

The nesting and ordering of the layout operators in a query correspond directly to the nesting and ordering of the data structures that they represent. This means that we can reorder or transform operators in the query to restructure the layout. Castor supports the following data structures, each of which has a corresponding layout operator:

- **Scalars:** Scalars can be integers, strings, Booleans, and decimal fixed-points.

- **Tuples:** Tuples are layouts that contain layouts of different types. If a collection contains tuples, all the tuples must have the same number of elements and their elements must have the same type. Tuples can be read either by taking the cross product or concatenating their sub-layouts.

- **Lists:** Lists are variable-length layouts. Their contents must be of the same type.

- **Hash indexes:** Hash indexes are mappings between scalar keys and layouts, stored as hash tables. Like lists, their keys must have the same type and their values must have the same type.

- **Ordered indexes:** Ordered indexes are ordered mappings between scalar keys and layouts.

At runtime, the layout operators read data from the layout and convert it into a relational form that the relational operators can consume. In Section 2.6.3, we discuss how these operators are implemented as iterators and how the iterators are composed together to form an executable query.

2.3.4 Staging

Another way to view the three semantic interpretations is from the point of view of multi-stage programming: the layout is constructed in the compile-time stage, and the compiled query reads the layout and processes it in the run-time stage. While traditionally program staging is used to implement code specialization, in the layout algebra, staging is used to implement data specialization. This difference in focus leads to different implementation challenges. In particular, the "unstaged" version of a layout-algebra program is often large (tens to hundreds of megabytes). The CASTOR compiler must be carefully designed to handle this scale.

We are particularly interested in layout-algebra programs that can be separated into a compile-time stage that constructs a layout and a runtime stage that reads it. Only a subset of layout-algebra programs can be separated in this way. We say that programs which have this property are *well-staged.*

At a high level, a program is well-staged if there are no compile-time dependencies on run-time data or vice versa. To formalize this intuition, we introduce run- and compile-time contexts. An expression is in a compile-time context if it appears in the first argument to `list`, `hash-idx`, `ordered-idx`, or `scalar`. Otherwise, it is in a run-time context. Additionally, the fields of relations are considered compile-time only and query parameters are run-time only. A program is well-staged if and only if the names referred to in compile-time contexts are bound in compile-time contexts and the names referred to in run-time contexts are bound in run-time contexts. The compiler uses a simple type system that tracks the stage of each name in the program to check well-stagedness.

Transforming a program into a well-staged form is a key goal of our automatic optimizer (Section 2.5). Many of the rules that the optimizer applies can be seen as moving parts of the query between stages.

2.4 Transformations

In this section, we define semantics-preserving transformation rules that optimize query and layout performance. These rules change the behavior of the program with respect to the layout and runtime semantics while preserving it with respect to the relational semantics. These rules subsume standard query optimizations because in addition to changing the structure of the query, they can also change the structure of the data that the query processes.

2.4.1 Notation

Transformations are written as inference rules. When writing inference rules, e will refer to scalar expressions, and q will refer to layout-algebra expressions. E and Q will refer to lists of expressions and layouts. In general, the names we use correspond to those used in the syntax description (Figure 2-5). If we need to refer to a piece of concrete syntax, it will be formatted as e.g., `concat` or `x`.

To avoid writing many trivial inductive rules, we define contexts (Figure 2-7) [34].

$$
\begin{aligned}
S &::= \{x_1 \mapsto e_1, \ldots, x_m \mapsto e_m\} \quad T ::= [q_1, \ldots, C, \ldots, q_n] \\
C &::= [\cdot] \mid \texttt{select}(S,\ C) \mid \texttt{filter}(e,\ C) \mid \texttt{join}(e,\ q,\ C) \mid \texttt{group-by}(S,\ E,\ C) \\
&\mid \ \texttt{dedup}(C) \mid \texttt{list}(q \texttt{ as } x,\ C) \mid \texttt{list}(C \texttt{ as } x,\ q) \mid \texttt{tuple}_\tau(T) \\
&\mid \ \texttt{hash-idx}(C \texttt{ as } x,\ q_v,\ e_k) \mid \texttt{hash-idx}(q_k \texttt{ as } x,\ C,\ e_k) \\
&\mid \ \texttt{ordered-idx}(C \texttt{ as } x,\ q_v,\ e_{lo},\ e_{hi}) \mid \texttt{ordered-idx}(q_k \texttt{ as } x,\ C,\ e_{lo},\ e_{hi})
\end{aligned}
$$

Figure 2-7: The grammar of contexts.

If C is a context and q is a layout algebra expression, then $C[q]$ is the expression obtained by substituting q into the hole in C. In addition to contexts, we define two operators: $\xrightarrow{t}$ and $\rightarrow$. $q \xrightarrow{t} q'$ means that the layout-algebra expression q can be transformed into q', and $q \rightarrow q'$ means that q can be transformed into q' in any context. The relationship between these two operators is:

$$q \rightarrow q' \equiv \forall C.\ C[q] \xrightarrow{t} C[q']$$

2.4.2 Relational Optimization

There is a broad class of query transformations that have been developed in the query-optimization literature [55, 14]. These transformations can generally be applied directly in Castor, at least to the relational operators. For example, commuting and reassociating joins, filter pushing and hoisting, and splitting and merging filter and join predicates are implemented in Castor. Although producing optimal relational-algebra implementations of a query is explicitly a non goal of Castor, these kinds of transformations are important for exposing layout optimizations.

2.4.3 Projection

Projection, or the removal of unnecessary fields from a query, is an important transformation because many queries only use a few fields; the most impactful layout specialization that can be performed for these queries is to remove unneeded fields.

First, we need to decide what fields are necessary. For a query q in some context C, the necessary fields in q are visible in the output of $C[q]$ or are referred to in C. Let $\textsc{free}(\cdot)$ be a function which returns the set of free variables in a context or layout expression. Let $\textsc{needed}(\cdot,\cdot)$ be a function from contexts C and layouts q to the set of necessary fields in the output of q:

$$\textsc{needed}(C,\ q) = \textsc{schema}(q) \cap (\textsc{schema}(C[q]) \cup \textsc{free}(C))$$

$\textsc{needed}(\cdot,\cdot)$ can be used to define transformations which remove unnecessary parts of a layout. For example, this rule removes unnecessary fields from tuples:

$$\frac{Q' = [q \mid q \in Q,\ \textsc{needed}(C,\ q) \neq \emptyset]}{C[\texttt{tuple}_\tau(Q)] \xrightarrow{t} C[\texttt{tuple}_\tau(Q')]}$$

There is a similar rule for `select` and `group-by` operators.

The projection rules differ from the others in this section because they refer to the context C. The other rules can be applied in any context. The context is important for the projection rules because without it, all the fields in a layout would be visible and therefore “necessary”. Referring to the context allows us to determine which fields are visible to the user.

2.4.4 Precomputation

A simple transformation that can improve query performance is to compute and store the values of parameter-free terms. This transformation is similar to partial evaluation.

The following rule[5] precomputes a static layout-algebra expression:

$$\frac{\textsc{schema}(\boxed{q}) = [f_1, \ldots, f_k] \qquad x \text{ is fresh}}{\boxed{q} \to \texttt{list}(\boxed{q}\ \texttt{as}\ x,\ \texttt{tuple}_{\textsf{cross}}[\texttt{scalar}(x.f_1),\ \ldots,\ \texttt{scalar}(x.f_k)])}$$

Hoisting static expressions out of predicates can also be very profitable:

$$\frac{x,\ y \text{ are fresh} \qquad \boxed{e'} \text{ is a term in } e \qquad \textsc{free}(e') \cap \textsc{schema}(q) = \emptyset}{\texttt{filter}(e,\ q) \to \texttt{depjoin}(\texttt{scalar}(e' \mapsto y)\ \texttt{as}\ x,\ \texttt{filter}(e[e' := x.y],\ q))}.$$

The expression e' can be precomputed and stored instead of being recomputed for every invocation of the filter. Similar transformations can be applied to any operator that contains an expression. This rule is useful when the filter appears inside a layout operator. For example, in the query $\texttt{list}(q\ \texttt{as}\ x,\ \texttt{filter}(e,\ q'))$, a subexpression of e can be hoisted out of the filter if it refers to the fields in q but not the fields in q'.

In a similar vein, `select` operators can be partially precomputed. For example:

$$\frac{y' \text{ is fresh} \qquad q'_v = \texttt{select}(\{\texttt{sum}(e) \mapsto y'\},\ q_v)}{\begin{array}{c}\texttt{select}(\{\texttt{sum}(e) \mapsto y\},\ \texttt{ordered-idx}(q_k\ \texttt{as}\ x,\ q_v,\ t_{lo},\ t_{hi})) \to \\ \texttt{select}(\{\texttt{sum}(y') \mapsto y\},\ \texttt{ordered-idx}(q_k\ \texttt{as}\ x,\ q'_v,\ t_{lo},\ t_{hi}))\end{array}}.$$

After this transformation, the ordered index will contain partial sums which will be aggregated by the outer select. This rule is particularly useful when implementing grouping and filtering queries, because the filter can be replaced by an index and the aggregate applied to the contents of the index. A similar rule also applies to `select` and `list`. A simple version of this rule applies to `hash-idx`; in that case, the outer `select` is unnecessary.

This transformation is combined with group-by elimination (Section 2.4.5) in TPC-H query 1 to construct a layout that precomputes most of the aggregation.

[5]Some rules make a distinction for *parameter-free* expressions, which do not contain query parameters. In these rules, parameter-free expressions are denoted as $\boxed{e}$.

2.4.5 Partitioning

Partitioning is a fundamental layout transformation that splits one layout into many layouts based on the value of a field or expression. A partition of a relation r is defined by an expression e over the fields in r. Tuples in r are in the same partition if and only if evaluating e over their fields gives the same value.

To simplify the rules involving partitions, we define a function $\textsc{part}(\cdot,\ \cdot,\ \cdot)$ which takes a layout q, a partition expression e, and a name x, and returns a pair of queries q_k and q_v:

$$\textsc{part}(q,\ \boxed{e},\ x) = (q_k,\ q_v) = (\texttt{dedup}(\texttt{select}(\boxed{e},\ q)),\ \texttt{filter}(x.e = \boxed{e},\ q)).$$

In this definition, q_k evaluates to the unique valuations of e in r. These are the partition keys. Note that the expression q_v contains a free scope x. We use $x.e$ to denote the expression e with its names qualified by the scope x. Once $x.e$ is bound to a particular partition key, q_v evaluates to a relation containing only tuples in that partition.

The partition function is used to define rules that create hash and ordered indexes from filters:

$$\frac{x, n \text{ is fresh} \qquad \textsc{part}(q,\ \boxed{e},\ x) = (q_k,\ q_v) \qquad \textsc{free}(e') \cap \textsc{schema}(q) = \emptyset}{\texttt{filter}(\boxed{e} = e',\ q) \rightarrow \texttt{hash-idx}(q_k \texttt{ as } x,\ q_v,\ e')}$$

and

$$\frac{x \text{ is fresh} \quad \textsc{part}(q,\ \boxed{e},\ x) = (q_k,\ q_v) \quad (\textsc{free}(e_l) \cup \textsc{free}(e_h)) \cap \textsc{schema}(q) = \emptyset}{\texttt{filter}(e_l \leq \boxed{e} \wedge \boxed{e} \leq e_h,\ q) \rightarrow \texttt{ordered-idx}(q_k \texttt{ as } x,\ q_v,\ e_l,\ e_h)}.$$

Partitioning also leads immediately to a rule that eliminates $\texttt{group-by}(\cdot)$:

$$\frac{x \text{ is fresh} \qquad \textsc{part}(q,\ \boxed{K},\ x) = (q_k,\ q_v)}{\texttt{group-by}(E,\ \boxed{K},\ q) \rightarrow \texttt{list}(q_k \texttt{ as } x,\ \texttt{select}(E,\ q_v))}.$$

There is a slight abuse of notation in this rule. K is a list of expressions, so the filter in q_v must have an equality check for each expression in K. This group-by elimination rule is used in many of the TPC-H queries which contain `group-bys`.

2.4.6 Join Elimination

Partitioning can be used to implement join materialization: a powerful transformation that can significantly reduce the computation required to run a query, at the cost of increasing the size of the data that the query runs on. Joins are often the most expensive operations in a relational query, so choosing a good join-materialization strategy is critical. CASTOR's layout operators admit several options for join materialization.

For example a join can be materialized as a list of pairs:

$$\frac{x \text{ is fresh} \qquad \textsc{part}(q,\ \boxed{e},\ x) = (q_k,\ q_v) \qquad \textsc{part}(q',\ \boxed{e'},\ x) = (q'_k,\ q'_v) \qquad \textsc{free}(e) \subseteq \textsc{schema}(q) \qquad \textsc{free}(e') \subseteq \textsc{schema}(q')}{\texttt{join}(\boxed{e} = \boxed{e'},\ q,\ q') \rightarrow \texttt{list}(\texttt{join}(\boxed{e} = \boxed{e'},\ q_k,\ q'_k)\ \texttt{as}\ x,\ \texttt{tuple}_{\textsf{cross}}[q_v,\ q'_v])}.$$

Each pair in this layout contains the tuples that should join together from the left- and right-hand sides of the join.

Joins can also be materialized as nested lists:

$$\frac{x \text{ is fresh} \qquad \textsc{schema}(q) = [f_1, \ldots, f_n] \qquad F = \texttt{scalar}(f_1), \ldots, \texttt{scalar}(f_n) \qquad \textsc{free}(e) \subseteq \textsc{schema}(q) \qquad \textsc{free}(e') \subseteq \textsc{schema}(q')}{\texttt{join}(e = e',\ \boxed{q},\ q') \rightarrow \texttt{list}(\boxed{q}\ \texttt{as}\ x,\ \texttt{tuple}_{\textsf{cross}}[F,\ \texttt{filter}(x.e = e',\ q')])}.$$

This layout works well for one-to-many joins, because it only stores each row from the left-hand side of the join once, regardless of the number of matching rows on the right-hand side.

Or a join can be materialized using a hash table:

$$\frac{\begin{array}{c} x, x' \text{ are fresh} \qquad \textsc{part}(q',\ \boxed{e'},\ x') = (q_k,\ q_v) \\ \textsc{schema}(q) = [f_1, \ldots, f_n] \qquad \textsc{schema}(q') = [f'_1, \ldots, f'_m] \\ \textsc{free}(e) \subseteq \textsc{schema}(q) \qquad \textsc{free}(e') \subseteq \textsc{schema}(q') \end{array}}{\begin{array}{l} \texttt{join}(e = \boxed{e'},\ q,\ q') \rightarrow \\ \quad \texttt{depjoin}(q \texttt{ as } x,\ \texttt{select}([f_1, \ldots, f_n, f'_1, \ldots, f'_m], \\ \qquad\qquad\qquad\qquad \texttt{hash-idx}(q_k \texttt{ as } x',\ q_v,\ x.e))) \end{array}}.$$

This is similar to how a traditional database would implement a hash join, but in our case the hash table is precomputed. Using a hash table adds some overhead from the indirection and the hash function but avoids materializing the cross product if the join result is large.

If the join is many-to-many with an intermediate table, then either of the above one-to-many strategies can be applied.

2.4.7 Predicate Precomputation

In some queries, it is known in advance that a parameter will come from a restricted domain. If this parameter is used as part of a filter or join predicate, precomputing the result of running the predicate for the known parameter space can be profitable, particularly when the predicate is expensive to compute. Let p be a query parameter and D_p be the domain of values that p can assume.

$$\frac{\begin{array}{c} \textsc{params}(e) = \{p\} \qquad w_i = e[p := D_p[i]] \\ e' = \bigvee_i (w_i \wedge p = D_p[i]) \vee e \qquad q' = \texttt{select}([w_1, \ldots, w_{|D_p|}, \ldots],\ q) \end{array}}{\texttt{filter}(e,\ q) \rightarrow \texttt{filter}(e',\ q')}$$

This rule generates an expression w_i for each instantiation of the predicate with a value from D_p. The w_is are selected along with the original query q. When we later create a layout for q, the w_is will be stored alongside it. When the filter is executed, if the parameter p is in D_p, the "or" will short-circuit, and the original predicate will not run. However, this transformation is semantics-preserving even if D_p is underapproximate. If the query receives an unexpected parameter, then it executes the original predicate e. Note that in the revised predicate e', $p = D_P[i]$ can be computed once for each i, rather than once per invocation of the filter predicate.

We use this transformation on TPC-H queries 2 and 9 to eliminate expensive string comparisons.

2.4.8 String Interning

Consider a more complex example: implementing string interning. This transformation will replace scalars (they do not have to be strings, but strings are a common use case) with unique identifiers, store the mapping between the scalars and the unique IDs in a hash index, and replace the IDs with the corresponding scalars when the layout is read. This transformation is valuable when there are few distinct scalar values, because each distinct value will only be stored once.

$$\frac{\begin{array}{c} q_{kv} = \texttt{select}(\{\mathsf{idx}() \mapsto k, f\}, \texttt{dedup}(\texttt{select}(\{f\}, r))) \\ r' = \texttt{select}(\{k\}, \texttt{join}(f = r.f, q_{kv}, r)) \\ q_{list} = \texttt{list}(r' \texttt{ as } n, \texttt{scalar}(k)) \\ q_{idx} = \texttt{hash-idx}(\texttt{select}(\{k \mapsto k'\}, q_{kv}), \texttt{scalar}()((\texttt{select}(\{f\}, \texttt{filter}(k = k', q_{kv})))), k) \end{array}}{r \to \texttt{depjoin}(q_{list}, q_{idx})}$$

In this transformation, f is a field in a relation r. This transformation has four parts:

1. q_{kv} relates scalars and keys. The scalars are deduplicated before giving them a key. $\mathsf{idx}()$ is a special function that returns an auto-incrementing integer.

2. r' is the relation r but with each instance of f replaced by its key.

3. q_{list} is a layout that contains the values in r'.

4. q_{idx} is a layout that contains the mapping between the keys and the scalars, stored as a hash index. When the tuple is scanned using the cross strategy, each

key in q_{list} is used to look up the correct value in q_{idx}.

2.4.9 Range Splitting

One interesting example of a data-dependent transformation is the compression of integers. All of our collections are homogeneous: their elements must be of the same type. This means, for example, that if a list of integers contains one integer which requires 64 bits to store, all the integers will be stored in 64 bits. If most of the integers in the list are much smaller—byte-sized, say—this will waste space. If the data in the list is reordered, then the small integers will be stored using fewer bytes and the large integers using more:

$$\frac{q_{lt} = \mathsf{filter}(|e| < 127, q) \qquad q_{gt} = \mathsf{filter}(|e| > 127, q)}{\begin{array}{l}\texttt{list}(q \texttt{ as } x, \texttt{scalar}(e)) \rightarrow \\ \quad \texttt{tuple}_{\mathsf{concat}}[\texttt{list}(q_{lt} \texttt{ as } x, \texttt{scalar}(e)), \texttt{list}(q_{gt} \texttt{ as } x, \texttt{scalar}(e))]\end{array}}.$$

2.4.10 Range Compression

We can make range splitting more effective by recognizing cases where values fall into a small range:

$$\frac{\begin{array}{c} min = \min_q e \\ q' = \mathsf{select}(\{(e' + min) \mapsto e\}, \texttt{scalar}((e - min) \mapsto e'))\end{array}}{\texttt{list}(q \texttt{ as } x, \texttt{scalar}(e)) \rightarrow \texttt{list}(q \texttt{ as } x, q')}.$$

Rewriting the values could allow us to use a smaller integer representation or to apply the previous transformation. Note that this transformation depends on the particular values stored in the layout. Castor can efficiently access the data for a layout expression by generating a SQL query and using an existing database system to execute it. We use the same mechanism when serializing a layout.

2.4.11 Correctness

To show that the semantics that we have outlined in Section 2.3.2 are sufficient to prove the correctness of nontrivial transformations, we prove the correctness of the equality-filter elimination rule (Section 2.4.5). Although we do not prove the correctness of all the rules, this example demonstrates that such proofs are possible.

In particular, since our notation mixes relational and layout constructs, even transformations that manipulate both the run- and compile-time behavior of the query are often local transformations and are therefore simple to prove correct.

We say that two programs q and q' are equivalent if they produce the same value in every context. We denote equivalence as $q \equiv q'$ according to the following rule:

$$\text{Equiv}\frac{\forall \sigma, \delta, s.\ \sigma, \delta \vdash q \Downarrow s \iff \sigma, \delta \vdash q' \Downarrow s}{q \equiv q'}$$

We say that a rule $q \to q'$ is semantics-preserving if $q \equiv q'$.

Now we prove that the filter-elimination rule:

$$\frac{x, n \text{ is fresh} \qquad \text{PART}(q,\ \boxed{e},\ x) = (q_k,\ q_v) \qquad \text{FREE}(e') \cap \text{SCHEMA}(q) = \emptyset}{\texttt{filter}(\boxed{e} = e',\ q) \to \texttt{hash-idx}(q_k\ \texttt{as}\ x,\ q_v,\ e')}$$

is semantics-preserving.

Theorem 1. *If* $\text{PART}(q,\ e,\ x) = (q_k,\ q_v)$ *and* x *is a fresh scope, then*

$$\texttt{filter}(e = e',\ q) \equiv \texttt{hash-idx}(q_k\ \texttt{as}\ x,\ q_v,\ e').$$

Proof. By Equiv, the right-hand side of this implication is equivalent to:

$$\forall \sigma, \delta, s.\ \sigma, \delta \vdash \texttt{filter}(e = e',\ q) \Downarrow s \iff \sigma, \delta \vdash \texttt{hash-idx}(q_k\ \texttt{as}\ x,\ q_v,\ e') \Downarrow s.$$

By R-HI,

$$\sigma, \delta \vdash \texttt{filter}(e = v,\ q) \Downarrow s \iff \sigma, \delta \vdash \texttt{depjoin}(q_k\ \texttt{as}\ x,\ \texttt{filter}(x.e = v,\ q_v)) \Downarrow s,$$

where $\sigma,\ \delta \vdash e' \Downarrow v$.

By the definition of PARTITION, $q_k = \texttt{dedup}(\texttt{select}(\{e\},\ q))$ and $q_v = \texttt{filter}(x.e = v,\ q)$, so

$$\begin{aligned}&\sigma, \delta \vdash \texttt{filter}(e = v,\ q) \Downarrow s \iff \\ &\quad \sigma, \delta \vdash \texttt{depjoin}(\texttt{dedup}(\texttt{select}(\{e\},\ q))\ \texttt{as}\ x,\ \texttt{filter}(x.e = v,\ \texttt{filter}(x.e = e,\ q))) \Downarrow s.\end{aligned}$$

We simplify the filter operators to get:

$$\begin{aligned}&\sigma, \delta \vdash \texttt{filter}(e = v,\ q) \Downarrow s \iff \\ &\quad \sigma, \delta \vdash \texttt{depjoin}(\texttt{dedup}(\texttt{select}(\{e\},\ q))\ \texttt{as}\ x,\ \texttt{filter}(x.e = v \land x.e = e,\ q)) \Downarrow s.\end{aligned}$$

Proving the correctness of this simplification is straightforward and does not rely on the correctness of the hash-index introduction rule.

By R-Filter and R-Depjoin (and some abuse of notation), this is equivalent to:

$$[t \mid t \leftarrow \texttt{filter}(e = v,\ q)] = \left[t \;\middle|\; \begin{array}{c} t' \leftarrow \texttt{dedup}(\texttt{select}(\{e\},\ q)) \\ t \leftarrow \texttt{filter}(t' = v \land t' = e,\ q) \end{array} \right].$$

At this point there are two cases of interest. First, assume that $v \in \texttt{dedup}(\texttt{select}(\{e\},\ q))$. By the semantics of `dedup`, v will appear exactly once in this query result if it appears at all. We can conclude that in this case:

$$\begin{aligned}&[t \mid t' \leftarrow \texttt{dedup}(\texttt{select}(\{e\},\ q)),\ t \leftarrow \texttt{filter}(t' = v \land t' = e,\ q)] \\ &\quad = [t \mid t' = v,\ t \leftarrow \texttt{filter}(t' = v \land t' = e,\ q)] \mathbin{+\!\!+} \\ &\qquad [t \mid t' \leftarrow \texttt{dedup}(\texttt{select}(\{e\},\ q)),\ t' \neq v,\ t \leftarrow \texttt{filter}(v = t' \land t' = e,\ q)] \\ &\quad = [t \mid t \leftarrow \texttt{filter}(v = v \land v = e,\ q)] \mathbin{+\!\!+} [t \mid t \leftarrow \texttt{filter}(v \neq v \land v \neq e,\ q)] \\ &\quad = [t \mid t \leftarrow \texttt{filter}(v = e,\ q)] \mathbin{+\!\!+} [\,] \\ &\quad = [t \mid t \leftarrow \texttt{filter}(v = e,\ q)].\end{aligned}$$

In the second case, assume that $v \notin \texttt{dedup}(\texttt{select}(\{e\},\ q))$. In this case:

$$\begin{aligned}
&[t \mid t' \leftarrow \texttt{dedup}(\texttt{select}(\{e\},\ q)),\ t \leftarrow \texttt{filter}(t' = v \land t' = e,\ q)] \\
&\quad = [t \mid t' \leftarrow \texttt{dedup}(\texttt{select}(\{e\},\ q)),\ t' \neq v,\ t \leftarrow \texttt{filter}(v = t' \land t' = e,\ q)] \\
&\quad = [t \mid t \leftarrow \texttt{filter}(v \neq v \land v \neq e,\ q)] \\
&\quad = [\,].
\end{aligned}$$

By our assumption, there is no e such that $e = v$, so $\texttt{filter}(e = v,\ q) = [\,]$.

In both cases, the two programs are equivalent, so we can conclude that the rule is semantics-preserving. □

2.5 Optimization

Castor includes an automatic, cost-guided optimizer for the layout algebra. The goal of the optimizer is to produce a *transformation sequence*, which is a sequence of transformations that (1) makes the query well-staged (Section 2.3.4) and (2) minimizes the cost of executing the query. The optimizer consists of two components: a transformation-scheduling language and a cost model for the layout algebra.

2.5.1 Scheduling

The space of transformation sequences is far too large for an exhaustive search. Instead, we consider a restricted space of sequences. We implemented a small domain-specific language—the *scheduling language*—that describes a search space of transformation sequences. This language is inspired by Stratego [106] and provides combinators for sequencing transformations, fix-points, selecting locations to apply transformations, and branching. Running a program in the scheduling language performs a search over transformation sequences. The optimizer is implemented as a program in the scheduling language, and it captures some of the domain knowledge that we have about how to optimize query layouts.

The optimizer scheduling program has four phases: join-nest elimination, hash-index introduction, ordered-index introduction, and precomputation. Cleanup transformations and other manipulations that allow the main transformations to apply are interleaved between these phases.

The join-nest elimination phase looks for unparameterized join nests and replaces them with layouts. As discussed in Section 2.4.6, there are several ways to eliminate a join operator. The right choice depends on whether the join is one-to-one or one-to-many. To eliminate a join nest, the optimizer performs an exhaustive search using the join-elimination rules and uses the cost model to choose the least expensive candidate.

The hash- and ordered-index introduction phases attempt to replace filter operators with indexes. When replacing a filter operator with an index, the most important choice to make is where in the query to place the filter. This choice determines which part of the layout the index will partition. The optimizer exhaustively searches over the possible index placements and uses the cost model to select the best candidate.

Finally, the precomputation phase selects unparameterized parts of the query to be computed and stored.

Before returning the query, the scheduler checks that it is well-staged (Section 2.3.4). If it is not, the query is discarded and scheduling fails.

We run the optimizer scheduling program in a Markov chain Monte Carlo outer loop that randomly disables transformations. A single run of the optimizer consists of many runs of the scheduling program, with transformations randomly disabled. We use the cost model to decide when to transition in the Markov chain. We keep track of the best transformation sequence that we have found and return that at the end of optimization.

The output of the optimizer is a sequence of transformation rules that introduce layout operators, minimizing the cost of executing the resulting query. A pleasant feature of the optimizer is that because it simply runs transformation rules, it is semantics-preserving if all the rules are. This means that all sequences are equally correct—they differ only in the quality of their results.

$$
\begin{aligned}
n &::= \mathbb{Z} \quad r ::= [n,\ n] \\
t &::= \mathsf{intT}(r) \mid \mathsf{boolT} \mid \mathsf{fixedT}(r,\ n_{scale}) \mid \mathsf{stringT}(r_{chars}) \mid \mathsf{tupleT}([t_1,\ \ldots,\ t_k]) \\
&\mid\ \ \mathsf{listT}(t,\ r_{elems}) \mid \mathsf{hash\text{-}idxT}(t_k,\ t_v,\ r_{keys}) \mid \mathsf{ordered\text{-}idxT}(t_k,\ t_v,\ r_{keys}) \\
&\mid\ \ \mathsf{emptyT} \mid \mathsf{funcT}(t_1, \ldots, t_m)
\end{aligned}
$$

Figure 2-8: The syntax of layout shapes.

Manual Optimization

The scheduling language can also be used to write manual transformation scripts. The optimizer scheduling program is general-purpose and includes several rounds of backtracking search, but the scheduling language can easily represent straight-line sequences of transformations.

2.5.2 Cost Model

When optimizing a query, we care primarily about its runtime cost; we assume that any compile-time cost is acceptable. The staged nature of the layout algebra makes estimating the runtime cost of a query complicated because the runtime cost depends on the sizes of the data structures in the layout. To compute these sizes we execute the compile-time portion of the query.

While method of computing costs is expensive, it is accurate because the true size of the data structure is evaluated. In future work, we intend to explore the use of cardinality estimates, similar to those used in query optimizers. These estimates must be carefully validated to ensure accuracy, but they can be much faster than building a layout for each candidate.

To assist in this cost estimation we introduce an abstraction of the query that we call a *layout shape* (Figure 2-8). A layout shape is essentially an abstract domain for the layout portion of a query. We use an interval abstraction to track the range of integer and fixed-point values in the layout as well as the number of elements in collections like lists and indexes. Given a layout shape, we can use simple models of the costs of the runtime query operators to estimate the cost of executing the entire query.

Computing the layout shape is expensive because it involves running the compile-time portion of the query. We want to compute the costs of many layouts during optimization, so optimizing the shape computation is critical. We use two techniques during optimization to make the shape computation cheaper. First, we compute the shape on a sample of the database. Using a sample means that the shape may be underapproximate (the ranges in the sample shape will be smaller than in the true shape), but we have found that this is acceptable during optimization. Second, we compute the shape of nested layouts in parallel. For example, to compute the shape of this layout:

$$\begin{aligned}&\texttt{list}(\texttt{dedup}(\texttt{select}(\{id\},\ log))\ \textsf{as}\ a,\\ &\quad\texttt{list}(\texttt{filter}(a.id = id,\ log)\ \textsf{as}\ b,\ \texttt{scalar}(b.exit))),\end{aligned}$$

Castor will issue these SQL queries:

$$\textsf{select count(distinct } id\textsf{) as } x_1 \textsf{ from } log;$$

$$\begin{aligned}&\textsf{select } min(c) \textsf{ as } x_2,\ max(c) \textsf{ as } x_3\\ &\textsf{from (select count() as } c \textsf{ from } log \textsf{ group by } id\textsf{);}\end{aligned}$$

$$\textsf{select } min(exit) \textsf{ as } x_4,\ max(exit) \textsf{ as } x_5 \textsf{ from } log;$$

It uses the results to construct this shape:

$$\textsf{listT}(\textsf{listT}(\textsf{intT}([x_4,\ x_5]),\ [x_2,\ x_3]),\ [x_1,\ x_1]),$$

where $[x_4,\ x_5]$ is the domain of the values of the `scalar` and $[x_2,\ x_3]$ and $[x_1,\ x_1]$ are

$$\frac{\sigma \vdash \texttt{scalar}(e \mapsto n) \Downarrow x \qquad x \text{ is an integer}}{\sigma \vdash \texttt{scalar}(e \mapsto n) : \mathsf{intT}([x, x])} \qquad \frac{\begin{array}{c}\sigma \vdash q_1 : t_1, \ldots, \sigma \vdash q_k : t_k \\ t = \mathsf{tupleT}([t_1, \ldots, t_k], \tau)\end{array}}{\sigma \vdash \texttt{tuple}_\tau([q_1, \ldots, q_k]) : t}$$

$$\frac{\sigma \vdash q : t}{\sigma \vdash \texttt{filter}(e,\ q) : \mathsf{funcT}(t)} \qquad \frac{\sigma \vdash q_k \Downarrow r \qquad t_v = \bigsqcup_{\sigma' \in r, \sigma' \vdash q_v : t'} t'}{\texttt{list}(q_k \texttt{ as } n,\ q_v) : \mathsf{listT}(t_v,\ [|r|,\ |r|])}$$

$$\frac{t = \mathsf{intT}([l,\ h]) \qquad t' = \mathsf{intT}([l',\ h'])}{t \sqcup t' = \mathsf{intT}([\min(l,\ l'),\ \max(h,\ h')])}$$

Figure 2-9: Selected semantics of the shape-inference pass.

the domains of the `list` lengths. These queries can be run concurrently, and if one of the queries times out we can approximate its results while still being able to compute the rest of the shape.

2.6 Compilation

The result of running the optimizer or manually applying transformation rules is a program in the layout algebra. This program is still quite declarative, so there is a significant abstraction gap to cross before the program can be executed efficiently. Compilation of layout-algebra programs proceeds in three phases: data-structure specialization, serialization and code generation.

2.6.1 Layout Semantics

The layout semantics describe how the layout portion of the program is converted to a binary representation. We use the specialized data structure (Section 2.6.2) for each operator to determine exactly how the layout is serialized.

Each of the layout operators has a binary serialization format which is intended to (1) take up minimal space and (2) minimize the use of pointers to preserve data locality.

- An integer is stored using a variable number of bytes (1–8).

- A date is represented as the number of days since the epoch and stored as an integer.
- A Boolean is represented as an integer that is either 0 or 1.
- A fixed-point number is normalized to a fixed scale and stored as an integer.
- A string is length-prefixed and is not null-terminated.
- A tuple is stored as a length-prefixed concatenation of its child layouts.
- A list is stored as a byte length and an element count followed by the concatenation of its elements. Lists can be efficiently scanned through but not accessed randomly by index, because the elements may be variable-sized.
- A hash index is implemented using minimal perfect hashes that index into a table of value offsets.
- An ordered index contains an ordered table of keys and value offsets. Lookups are performed using a binary search on this table.

Serialization proceeds as described in Figure 2-10. Each layout operator is serialized by first serializing its children, then constructing the appropriate data structure. For example, to construct the layout for `list`(q_k `as` k, q_v), we first serialize the dependent relation q_v for each tuple in q_k. We concatenate the resulting layouts and prepend the header fields *count* and *length*, which contain the number of list items and the length of the list in bytes respectively. The query operators have trivial layout semantics that simply concatenate their child layouts.

2.6.2 Data-Structure Specialization

The first step in the Castor compiler is to generate specialized instances of the layout-algebra data structures. Each instance of a layout operator in a query gets a specialized data-structure implementation. We use the shape (Section 2.5.2) of the layout to guide the specialization process.

$$b : \textit{Byte string} \quad \sigma, t : \textit{Tuple} \quad \delta : \textit{Id} \mapsto \textit{Relation}$$

$$\frac{}{\sigma, \delta \vdash \emptyset \downarrow \text{""}} \qquad \frac{\sigma,\ \delta \vdash q \downarrow b}{\sigma,\ \delta \vdash \texttt{filter}(e,\ q) \downarrow b} \qquad \frac{\sigma,\ \delta \vdash q \downarrow b \quad \sigma,\ \delta \vdash q' \downarrow b'}{\sigma,\ \delta \vdash \texttt{join}(e,\ q,\ q') \downarrow bb'}$$

$$\frac{\sigma,\ \delta \vdash e \Downarrow v \quad b \text{ is the binary format of } v}{\sigma,\ \delta \vdash \texttt{scalar}(e) \downarrow b}$$

$$\frac{\begin{array}{c}\sigma,\ \delta \vdash q_k \Downarrow [t_1, \dots, t_n] \qquad \forall 1 \le i \le n.\ \sigma \cup t_i,\ \delta \vdash q_v \downarrow b_i \\ \sigma,\ \delta \vdash \texttt{scalar}(|b_1| + \cdots + |b_n|) \downarrow b_{len} \quad \sigma,\ \delta \vdash \texttt{scalar}(n) \downarrow b_{ct}\end{array}}{\sigma,\ \delta \vdash \texttt{list}(q_k, q_v) \downarrow b_{ct} b_{len} b_1 \dots b_n}$$

$$\frac{\forall 1 \le i \le n.\ \sigma,\ \delta \vdash q_i \downarrow b_i \quad \sigma,\ \delta \vdash \texttt{scalar}(|b_1| + \cdots + |b_n|) \downarrow b_{len}}{\sigma,\ \delta \vdash \texttt{tuple}_\tau([q_1, \dots, q_n]) \downarrow b_{len} b_1 \dots b_n}$$

Figure 2-10: Selected layout semantics.

First, we compute the layout shape of the query. Unlike the optimizer, which uses a fast underapproximate method to compute the shape, the compiler needs an overapproximation. Computing an overapproximate shape ensures that if we construct a data structure that supports the values in the shape, it will support all the values in the true dataset.

Our data-structure implementations follow a common pattern. Each structure consists of a header that contains fields like lengths, counts, or offsets and a body that contains data. For example, strings are represented using a length header and a body of byte-encoded data. Castor supports the following generic specializations for all structures:

- *Field narrowing:* Castor uses the layout shape to determine the range of certain header fields (like string lengths). It then chooses the smallest byte-width that is large enough to support the range of values in the field.

- *Field elision:* If the layout shape shows that a field will only ever contain a single value, then Castor elides the field entirely. For example, this specialization allows fixed-size tuples to be stored with zero overhead.

Castor implements additional specializations for certain layout operators.

Integers are narrowed in the same way that fields are. We use the layout shape to determine the minimal number of bytes necessary to hold the values that appear in the layout. The narrowing can differ for each integer layout operator.

For fixed-point numbers, the layout shape contains a minimal scaling factor that is precise enough for all values and a domain covering the numerators of the fixed points. The numerators are stored as integers and the shared scaling factor is not stored. Each fixed-point layout operator can have a different scaling factor.

Hash indexes are implemented using a minimal perfect hash [9, 27] that is computed per-index. Using a minimal perfect hash allows our hash indexes to have high load factors (up to 99%) and also allows us to ignore the possibility of collisions.

After performing data-structure specialization, we serialize the layout as described in Section 2.6.1, using the specializations to decide which fields to narrow or remove.

2.6.3 Runtime Semantics

The last step in compilation is to generate the code that reads the layout. Each operator has a corresponding iterator: the layout operators read from the layout and produce streams of tuples and the relational operators consume and produce tuple streams. We construct these iterators according to the method in [101]. This method is referred to as push-based or data-centric query evaluation.

In push-based evaluation, each iterator takes a callback which it calls for each tuple in its output stream. To run the query, the user passes the root iterator a callback function that processes the output tuples. For a given query, the callbacks are known statically, so we inline them. This produces a single loop nest with no function calls and minimal branching.

Each operator in the layout algebra has a corresponding iterator implementation. The relational operators are implemented as described in [101]. The layout operators each have an iterator that reads the layout. These iterators are modified depending on the data-structure specializations being performed. Specifically, the iterators can be modified to implement the field-narrowing and elision specializations discussed in Section 2.6.2.

The push-based evaluation method is in contrast to pull-based evaluation, which is used in the traditional iterator model. In pull-based evaluation, each query operator is compiled to an iterator that consists of a state structure and a step function. Each time a query operator is stepped, it recursively steps its children and updates its state. The query is executed by repeatedly stepping the root iterator. We implemented pull-based evaluation in an early prototype of Castor and found that optimizing the resulting code was difficult because of the large amount of branching and control flow.

The drawback of push-based query evaluation is that certain operators, such as deduplication and ordering, must buffer their inputs before processing them. Rather than implement buffering, we restrict the use of these operators and wherever possible we replace them with layout-based implementations that perform these operations at compile time.

Code Generation

Castor performs code generation in two phases. First, a syntax-directed lowering pass transforms each query and layout operator into an imperative intermediate representation, using the layout shape to generate the layout-reading code. Next, we run loop-invariant code motion and reorder the evaluation of predicates so that expensive clauses in conjunctive predicates are evaluated last. Finally, we lower the imperative IR to LLVM IR. LLVM performs further low-level optimizations and emits code, completing the code-generation phase.

2.6.4 Performance of Staging

As discussed in Section 2.5.2, running the compile-time part of a query is costly. During compilation of a query, we run its compile-time part once to generate a layout shape and again during serialization. We evaluate the compile-time part of a query by translating it to a SQL query, running that query against a backing database, then folding over the results. This process is the most expensive part of compilation, since it involves executing a complex query and reading a large amount of data from the database. Unlike the optimizer, the compiler cannot tolerate an approximate result, so we implemented two optimizations that do not involve approximation to manage this cost.

First, we perform a series of query optimizations before sending a query to the database. The most important optimization is the removal of dependent joins [74]. Each level of layout nesting introduces an additional dependent join, and many database optimizers are not able to optimize them away. Removing them significantly improved the performance of our queries.

Second, we use Amazon Redshift as the backing database for the Castor compiler. Redshift is a high-performance distributed column-oriented database, and switching to Redshift from PostgreSQL yielded a significant performance improvement. Castor is not a good fit for this use case, because the queries that the compiler generates are specific to the program being compiled, so are not amenable to caching.

After implementing these optimizations, the compilation time for TPC-H query 3 dropped from over 12 hours to less than 20 minutes. Running the compiler using a two-node Redshift cluster, the maximum compilation time for an optimizer generated query is 100 minutes with a median of 4.8 minutes.

2.7 Evaluation

We test three hypotheses about Castor's performance: (1) Castor's layout optimizations produce queries that are faster than existing in-memory databases, (2) the resulting layouts are smaller than in existing databases, and (3) the deductive approach to query optimization scales better than generate-and-test approaches.

We compare Castor with three other systems: Hyper [73], Cozy [66], and Chestnut [111]. Hyper is an in-memory column-store which has a state-of-the art vectorizing query compiler. It implements compilation techniques that are well outside the scope of this paper, such as using SIMD operations to operate on multiple tuples at a time. Cozy is a state-of-the-art generate-and-test-based program-synthesis tool that generates specialized data structures from relational queries. Chestnut is similar to Cozy but focuses on object queries.

2.7.1 TPC-H Analytics Benchmark

In Section 2.2, we evaluated Castor on a query from the DemoMatch system. In this section, we perform an in-depth evaluation on the TPC-H benchmark. TPC-H is a standard database benchmark, focusing on analytics queries. It consists of a data generator, 22 query templates, and a query generator which instantiates the templates. The queries in TPC-H are inherently parametric, and their parameters come from the domains defined by the query generator. To build our benchmark, we took the query templates from TPC-H and encoded them as Castor programs. It is important that the queries be parametric. Specializing a nonparametric query is uninteresting because it can simply be evaluated and the result stored.

TPC-H is a general-purpose benchmark, so it exercises a variety of SQL primitives. We chose not to implement all of these primitives in Castor, not because they would be prohibitively difficult, but because they are not directly related to the layout-specialization problem. In particular, Castor does not support executing `order-by`, `group-by`, `join`, or `dedup` operators at runtime[6], and it does not support limit clauses at all. Some of these operators can be replaced by layout specialization, but others cannot. We implemented all the queries in TPC-H, except for query 13 because it contains an outer join. We removed runtime ordering and limit clauses from seven other queries. When evaluating the TPC-H queries, we used the 1GB scale factor. We ran our benchmarks on an Intel Xeon W–2155 with 64GB of memory.

2.7.2 Comparison with Hyper

We run each benchmark query using the following configurations:

- *Baseline:* We use Hyper as our baseline. Each query is run on a database containing the full TPC-H dataset.

[6]These operators can be processed into the compiled form of the query.

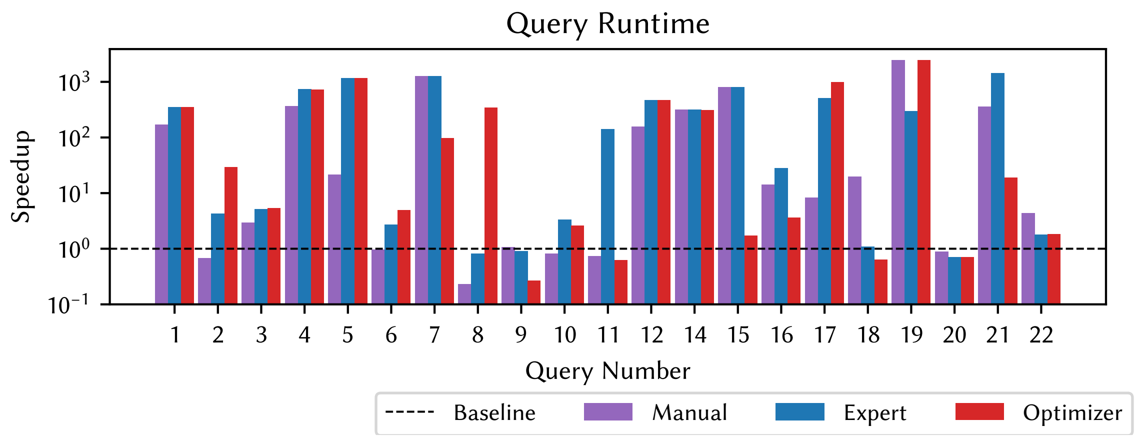


Figure 2-11: Performance on TPC-H queries.

- *Manual:* We manually implement the transformations that Castor performs in Hyper. We do this by generating a specialized set of materialized views and indexes that replicate the specialized layout the Castor produces. Hyper supports a smaller space of layouts than Castor, so this translation is best-effort. Hyper does not support nested layouts, for example.

- *Expert:* We run Castor using an expert-written transformation sequence for each query which generates an efficient, well-staged version of the query. The advantages over the manual approach are: (1) the transformations are correctness-preserving, so we don't have to worry about introducing bugs while optimizing and (2) we can use the Castor compiler to generate the specialized layout and query code.

- *Optimizer:* We run the Castor optimizer (Section 2.5) on a direct translation from the SQL implementation of the query to the layout algebra. The optimizer searches over the space of transformation sequences, using its cost model to select the best sequence. We run the optimizer with a two-hour timeout.

For each query and configuration, we measure its runtime, layout size, and memory footprint.

Runtime

Figure 2-11 shows the speedup over baseline Hyper for the *Manual*, *Expert*, and *Optimizer* configurations. The *Manual* configuration requires the most work from the user and offers no assurance of correctness. It is faster than the baseline more than half of the time. The *Expert* configuration is faster than the baseline 85% of the time and faster than the *Manual* configuration 60% of the time. The optimizer beats the baseline 80% of the time.

Of the 20% of queries (9, 11, 18, and 20) where the optimizer does not beat the baseline, only query 9 is significantly (more than 2x) slower than baseline Hyper. Queries 9, 11, and 18 have Expert configurations that match or exceed Hyper's performance, which suggests that the deficiency is in the optimizer, not in Castor's compiler or runtime.

These results show that the *Expert* and *Optimizer* configurations offer a compelling performance advantage over the baseline and a compelling user-interface advantage over the *Manual* configuration.

Layout Size

We recorded the size of the layouts for the *Manual*, *Expert*, and *Optimizer* configurations. We exclude *Baseline* because it has a large constant size for all queries (approximately the size of the TPC-H data—1GB). Figure 2-12 shows *Expert* and *Optimizer* with *Manual* as the new baseline. *Expert* beats *Manual* on all benchmarks, and *Optimizer* beats *Manual* on 95%.

The absolute size of the *Expert* layouts is small—less than 10MB for 55% of the queries and less than 100MB for 90% of the queries. The size of all layouts is 918.4MB, which compares favorably to the 1.1GB original data set. The size difference between Castor's layouts and the original data supports the hypothesis that parameterized queries rely on fairly small subsets of the whole database, making layout specialization a profitable optimization even when it involves replication.

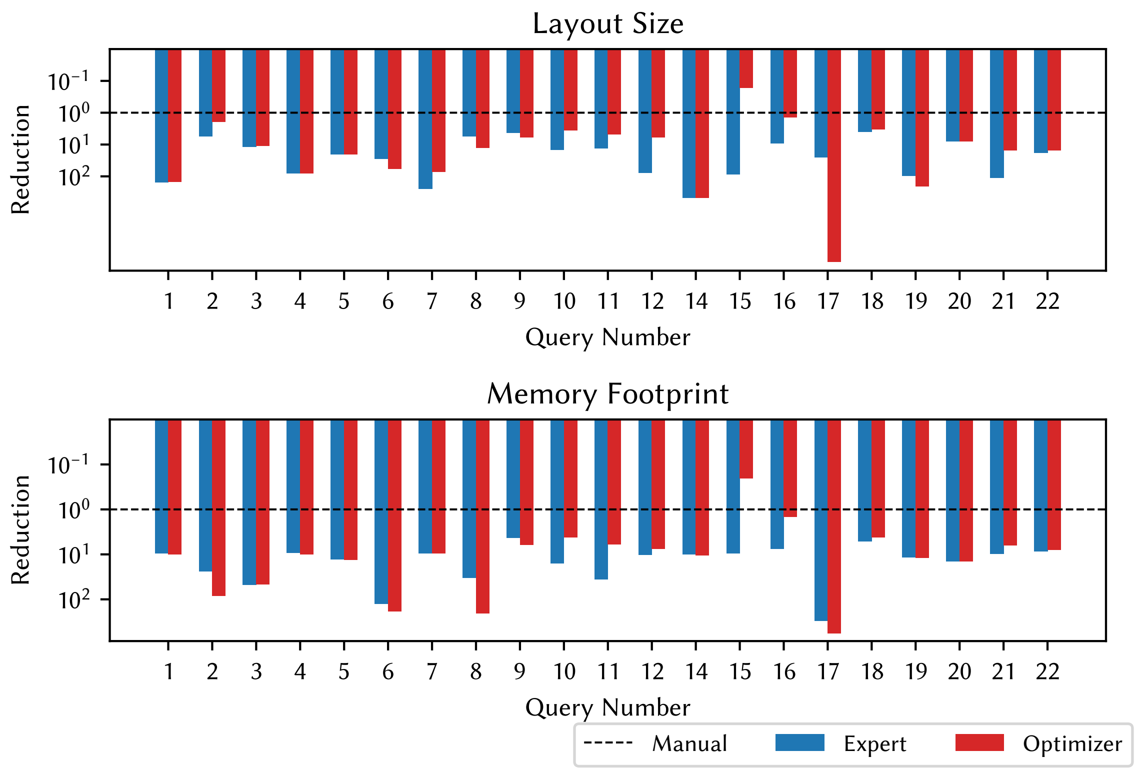


Figure 2-12: Layout size of TPC-H queries.

Memory Use

Finally, we measured the peak memory use of the query process for each query. HYPER consistently uses the same amount of memory as the layout size. In some cases it uses more, presumably because it has large runtime dependencies like LLVM. In contrast, Figure 2-12 shows that CASTOR's peak memory use is significantly lower than HYPER for all expert queries and for 95% of the optimizer queries.

2.7.3 Comparison with COZY & CHESTNUT

We compared CASTOR against two state-of-the-art data-structure-synthesis tools: COZY and CHESTNUT.

COZY

We transformed our input queries into COZY's specification format and ran COZY with a six-hour timeout. In this configuration, we found that COZY was unable to make significant improvements on all but two of the TPC-H queries. On Q4, Cozy precomputed one of the joins and a filter. On Q17, Cozy added an index. We ran both of these queries and found that despite the optimization Q4 was slower than baseline Hyper at 5.4s and Q17 was too slow to run on the entire TPC-H dataset.

There are two reasons why Castor performs better than Cozy in our comparison. First, Castor's deductive approach to optimization scales better than Cozy's generate-and-test method as the query size increases. This means that Castor is able to spend more time choosing between data structures, rather than looking for a correct implementation. Second, Castor has custom implementations of its data structures that take advantage of the fact that the database is read-only and known to the compiler. Cozy uses the C++ STL collections, which can't make either assumption. So, Castor is somewhat better at choosing layouts and has layouts that are better-tuned for its use case.

Chestnut

We attempted to use Chestnut to optimize four of the TPC-H queries, which were implemented as benchmarks by the Chestnut authors, but we were unable to build and run the generated code. Manually examining the layouts for Q1 and Q3–6, we find that Chestnut uses projection and indexes in many of the same places that Castor does but misses some optimizations that Castor can take advantage of, such as aggregate precomputation (Section 2.4.4). As Chestnut uses a generate-and-test strategy that is similar to Cozy, it is likely to have similar scalability problems on larger queries. Like Cozy, it relies on C++ collections which do not have data-specific specializations.

2.7.4 Performance of Multiple-Query Workloads

We experimented with combining pairs of queries using the following reduction. A pair of queries q and q' may be reduced to a single query with an additional parameter

Query Pair	Query	Runtime (ms)		Memory (MB)		Size (MB)	
		Opt.	*Man.*	*Opt.*	*Man.*	*Opt.*	*Man.*
1 & 2	1	0.02	0.03	2.3	233.9	102.9	206.6
	2	0.08	3.55	2.9	233.9		
2 & 3	2	0.08	3.57	2.9	1088.8	220.6	1088.4
	3	4.77	7.46	25.8	1089.1		
3 & 4	3	5.05	7.20	25.5	879.4	130.8	966.8
	4	0.08	0.02	3.4	879.5		
4 & 5	4	0.08	0.02	3.3	35.4	17.5	21.0
	5	0.02	0.54	2.9	35.5		

Table 2.1: Performance of pairs of queries compiled together. *Opt.* is the query produced by the optimizer and *Man.* is the query that we hand-optimized (by applying our equivalence-preserving rewriting rules).

id that chooses the query to execute:

$$\texttt{filter}(qid = id,\ \texttt{tuple}_{\mathsf{concat}}([\texttt{tuple}_{\mathsf{cross}}[\texttt{scalar}(0 \mapsto qid),\ q],$$
$$\texttt{tuple}_{\mathsf{cross}}[\texttt{scalar}(1 \mapsto qid),\ q']])).$$

While this forces the two queries to have the same schema, a unified schema can always be produced by taking the union of the fields emitted by the queries and returning null for the fields that do not correspond to the current query. The consumer of the query result can only examine the fields that correspond to the query that they need.

We find that the performance of the combined queries is no worse than the performance of the queries when compiled separately. However, our optimizer does not take advantage of sharing opportunities between multiple queries, so the size of the combined layout is the sum of the sizes of the individual layouts. Adding transformations that exploit sharing is left to future work.

2.7.5 Summary

Our evaluation shows that 95% of the time, Castor produces queries that are faster than a state-of-the-art in-memory database. The layout optimizations that Castor implements are able to outperform a heavily engineered vectorizing compiler, in some cases by multiple orders of magnitude. The layouts produced by Castor are up to two orders of magnitude smaller than a state-of-the-art in-memory database. This reduction in size translates to up to two orders of magnitude reduction in memory footprint. Finally, Castor's deductive optimization scales better than existing generate-and-test synthesis methods. It is able to optimize 21 of the TPC-H queries—more than the existing techniques—while supporting a rich space of optimizations.

2.8 Related Work

Deductive Synthesis. There is a long line of work that uses deductive synthesis and program-transformation rules to optimize programs [5, 83] and to build performance DSLs [84, 100]. Castor is a part of this line of work: it is a performance DSL which uses deduction rules to generate and optimize layouts. However, its focus on particular data sets and on deduction rules to optimize data in addition to programs separates it from previous work.

Deductive Data-Representation Synthesis Fiat [28] is a closely related system that applies deductive synthesis to generate efficient data structure implementations from high level relational specifications. Fiat is more general than Castor; it is based on a theory of abstract data type (ADT) refinement. Using these refinements, high-level, nondeterministic ADT specifications can be refined into low-level implementations. By default, this process is performed semi-manually; Fiat provides support for the refinement process, but some user input is required. Fiat's program representation is object-like, in contrast to Castor's relational-algebra language. This representation allows Fiat to express ADT methods that share data structures. It is our experience that Castor's simple program representation makes writing transformations

	Manual				*Expert*			*Optimizer*		
Q#	Time[3]	Time[2]	Mem.	Size	Time[2]	Mem.	Size	Time[2]	Mem.	Size
1	5.11	0.03	22.9	17.8	0.01	2.4	0.1	0.01	2.3	0.1
2 [0]	2.34	3.46	231.9	206.6	0.55	9.8	37.2	0.08	2.7	104.0
3 [01]	22.35	7.67	877.4	966.8	4.33	18.3	81.0	4.17	19.1	85.6
4	7.33	0.02	22.7	17.8	< 0.01	2.5	0.2	< 0.01	2.2	0.2
5	11.71	0.55	33.3	24.1	< 0.01	2.6	1.2	< 0.01	2.5	1.2
6	2.01	2.10	897.6	858.8	0.74	7.1	31.0	0.41	4.8	14.8
7	12.51	0.01	21.8	17.8	< 0.01	2.3	< 0.1	0.13	2.3	0.2
8	3.46	15.00	527.4	375.4	4.24	15.8	68.9	< 0.01	2.6	29.5
9	35.70	33.79	1544.9	1550.8	39.71	358.6	365.0	132.75	252.7	256.5
10 [01]	23.00	28.15	430.3	375.4	6.92	27.1	25.4	8.83	102.9	103.1
11	5.63	7.73	94.7	68.2	0.04	2.6	5.3	8.95	15.7	13.9
12	4.70	0.03	23.1	17.8	< 0.01	2.3	0.2	< 0.01	3.1	3.0
14	3.12	< 0.01	21.8	17.8	< 0.01	2.2	< 0.1	< 0.01	2.1	< 0.1
15	8.06	< 0.01	22.1	17.8	< 0.01	2.4	0.2	4.70	106.8	107.2
16 [1]	39.56	2.78	39.8	35.7	1.41	5.3	3.9	10.94	27.1	25.6
17	9.83	1.20	1224.9	1224.7	0.02	4.1	49.6	< 0.01	2.1	< 0.1
18 [0]	47.66	2.43	375.5	295.7	44.26	74.1	73.5	74.66	88.7	88.5
19	24.44	0.01	25.5	17.8	0.08	2.2	0.2	< 0.01	2.1	< 0.1
20	8.73	9.87	1300.9	1377.8	12.46	91.1	173.3	12.27	91.8	173.3
21 [01]	14.33	0.04	22.8	21.0	< 0.01	2.3	0.2	0.75	3.6	1.4
22 [1]	20.09	4.61	33.5	31.5	11.36	3.9	1.8	10.87	4.2	2.0

[0] Limit clause removed. [1] Run-time ordering removed. [2] Specialized. [3] Unspecialized.

Table 2.2: Runtime of queries derived from TPC-H (ms). Memory use is the peak resident set size during a query (MB). Size is the layout size (MB).

straightforward; it is unclear whether writing correct refinements for Fiat is more or less complex than writing transformations for CASTOR.

While Fiat can be used manually, a more automated approach is possible in domains that have well-understood design spaces. Delaware et al. apply Fiat to the data structure synthesis problem for SQL queries. They show that a specialized planning tactic can automate the refinement of a collection of queries into an efficient implementation. However, only a limited subset of SQL is considered, so more work would be required to apply Fiat to real-world query optimization problems

Inductive Data-Representation Synthesis. The layout-optimization problem is similar to the problem of synthesizing a data structure that corresponds to a relational specification [50, 51, 65, 66, 100].

The best data-structure-synthesis tool—COZY—uses a generate-and-test strategy. The testing phase uses an SMT solver to perform bounded verification of candidates. In our experiments (Section 2.7.2) we found that COZY's verification step does not scale to the TPC-H queries. CASTOR uses deductive synthesis to avoid this costly verification step by only searching the space of correct programs.

CHESTNUT is another tool for layout optimization [111]. It differs from CASTOR and COZY in that it considers object queries from ORMs rather than SQL. Like COZY, CHESTNUT relies on a generate and test strategy with a bounded verification step. Its optimizer exhaustively enumerates layouts and query plans separately and uses the verifier to determine if a plan/layout combination is correct. CHESTNUT avoids the scalability problems inherent to this approach by restricting its search space. For example, it only considers a single level of layout nesting, which limits data locality. It does not consider plans that perform partial aggregation (Section 2.4.4) or precomputation of predicates (Section 2.4.7). Any extensions to CHESTNUT must be carefully chosen to avoid search-space explosion. In contrast, CASTOR can support complex query transformations in a straightforward way because they do not need to be discovered by an exhaustive search.

Unlike the previous work, CASTOR considers a refinement of the data-structure-synthesis problem where both the query and the dataset are known to the compiler. This additional information allows Castor to use optimizations which would not be safe if the data were not known. In particular, Castor is able to significantly reduce the size of its optimized data sets by generating specialized collection implementations based on the properties of the data to be stored. The existing work relies on off-the-shelf collections libraries which cannot be specialized in this way.

Database Storage. Traditional databases are mostly row-based. Column-based database systems (e.g., Hyper [73], MonetDB [7] and C-Store [99]) are popular for OLAP applications, outperforming row-based approaches by orders of magnitude. However, the existing work on database storage generally considers specific storage optimizations (e.g., [1]) or specializations that benefit broad classes of data such as scientific [98] or geo-spatial [47] data. One exception is RodentStore [26], which proposed a language to describe storage layouts and showed that different layouts could benefit different applications. However, a compiler was never developed to create the layouts from this language; the paper demonstrated its point by implementing each layout by hand.

Materialized View and Index Selection. The layouts that Castor generates are similar to materialized views, in that they store query results. Castor also generates layouts which contain indexes. Several problems related to the use of materialized views and indexes have been studied (see [48] for a survey): (1) the view-storage problem that decides which views need to be materialized [19], (2) the view-selection problem that selects view(s) that can answer a given query, (3) the query-rewriting problem that rewrites the given query based on the selected view(s) [81], (4) the index-selection problem that selects an appropriate set of indexes for a query [97, 46, 10, 102]. However, materialized views are restricted to being flat relations. The layout space that Castor supports is much richer than that supported by materialized views and indexes. In addition, the view-selection literature has not previously considered the problem of generating execution plans for chosen views and indexes.

Query Compilation. Castor uses techniques from the query-compilation literature [101, 91, 58]. It extends these techniques by using information about the layout to specialize its queries.

2.9 Conclusion

We have presented Castor, a domain-specific language for expressing a wide variety of physical database designs, and a compiler for this language. We have evaluated it empirically and shown that it is competitive with the state of the art in memory database systems.

Chapter 3

Metric Program Synthesis

3.1 Introduction

Programming-by-example (PBE) is a program-synthesis task where the goal is to learn a program in some domain-specific language (DSL) that is consistent with a set of input-output examples. Because PBE can automate custom tasks without requiring programming skills, this topic has attracted enormous attention from several research communities and has found many useful applications ranging from string and table transformations [44, 35] to question answering [16] to computer-aided design (CAD) [31].

Techniques for solving the PBE problem can be classified as either *domain-agnostic* or *domain-specific*. Domain-specific methods (e.g., [44, 31, 35, 16, 15, 107, 38]) are specialized to DSLs and target pre-defined classes of synthesis tasks. Domain-agnostic methods [96, 36, 109] are parameterized over DSLs and can, in principle, be applied to a variety of domains.

A common domain-agnostic solution to the PBE problem is to perform *bottom-up enumeration* over DSL programs [104, 109, 69]. The idea is to start with primitives in the DSL and build up increasingly complex programs by combining existing terms via DSL constructs. The key challenge when scaling this approach to large programs is state explosion, since the number of programs grows exponentially with the search depth. Prior work has proposed several techniques to combat the state-explosion problem.

One simple but effective technique is to leverage *observational equivalence*: programs that produce the same output on the given set of input examples are effectively identical (at least for PBE purposes), so it suffices to keep only one representative program. For example, synthesis techniques based on *finite tree automata (FTA)* leverage observational equivalence to compactly represent of the set of all programs consistent with the given input-output examples.

Observational equivalence only reduces the state space in scenarios where many programs share the same (relevant) input-output behavior, but this property does not hold in all domains. Consider the *inverse constructive solid geometry (CSG) problem*, where the goal is to "decompile" a complex geometric shape into a set of geometric operations that were used to construct it in a computer-aided design (CAD) system [31, 110]. While this problem can be framed as a PBE task [31], observational equivalence only modestly reduces the search space because few programs produce *exactly* the same image. However, we notice that, in this domain, many programs have similar—but not identical—input-output behaviors. Furthermore, replacing a subprogram with one that has similar behavior often causes a small change to the overall program behavior, which suggests that these replacements can be repaired. This observation motivates the following question: can we relax the observational-equivalence criterion and develop a synthesis algorithm that exploits our domain knowledge about the *semantic similarity* between programs?

In this chapter, we answer this question affirmatively and present a new domain-agnostic PBE algorithm called *metric program synthesis*. We generalize observational equivalence to a weaker criterion called *observational similarity* by replacing the equivalence relation with an expert-provided distance function δ. Our method clusters programs into the same equivalence class if their output is within a radius ϵ according to δ. This distance metric gives DSL designers a powerful way to inject domain knowledge into the search without building an entirely domain-specific algorithm.

To exploit observational similarity, our metric-program-synthesis approach proceeds in two phases: First, it performs bottom-up enumerative synthesis to build a *version space* [61] that represents all programs up to some fixed AST depth. During bottom-up enumeration, it clusters programs into equivalence classes using the provided distance metric, keeping one representative of each equivalence class. Distance-based clustering introduces approximation: the version space contains many programs that are incorrect but *close to* being correct. We introduce a second *local search* step to repair these incorrect programs: starting with a program P whose output is close to the goal, our technique performs hill-climbing search guided by δ to find a syntactic perturbation P' of P that has the intended input-output behavior. This second local-search step mitigates the incompleteness introduced by distance-based clustering and allows recovering programs that are not explicitly contained in the reduced search space.

We implemented our approach in a tool called SyMetric and evaluate it on three domains: (1) inverse CSG [31], (2) regular-expression synthesis [62, 15], and (3) tower building [33]. Our evaluation shows that, when provided with appropriate distance functions, SyMetric is competitive with synthesizers designed/trained for these domains, and it outperforms other domain-agnostic synthesizers that use observational equivalence.

3.1.1 Contributions

To summarize, this chapter makes the following contributions:

- We introduce *metric program synthesis* as a way to generalize observational equivalence and give DSL designers a new mechanism for injecting domain knowledge into the synthesizer.
- We show how to use distance metrics to perform effective clustering, ranking, and repair of programs explored during synthesis.
- We evaluate our implementation, SyMetric, in three application domains and compare it against several relevant baselines.

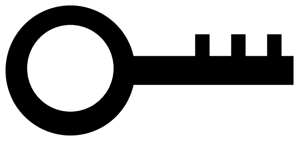


Figure 3-1: The input image.

3.2 Overview

In this section, we work through an example that illustrates the key ideas in our algorithm.

Consider the picture of a key shown in fig. 3-1. To a human, it is clear that this image contains three important pieces: circles to make up the handle of the key, a rectangle for the shaft, and evenly spaced rectangles for the teeth. We can write a program that generates this key in a simple DSL that includes primitive circles and rectangles as well as union, difference, and repetition operators:

$$(\texttt{Circle}(4,8,4)-\texttt{Circle}(4,8,3))\cup\texttt{Rect}(7,7,15,9)\cup\texttt{Repeat}(\texttt{Rect}(10,9,11,10),2,0,3)$$

The program composes the three shapes: a hollow circle for the handle of the key, a rectangle for the shaft, and three small, evenly spaced rectangles for the teeth. The hollow circle is constructed by subtracting a small circle from a larger one: $\texttt{Circle}(x,y,r) - \texttt{Circle}(x',y',r')$. The evenly spaced teeth are constructed by replicating a small rectangle three times: $\texttt{Repeat}(\texttt{Rect}(x,y,x',y'),dx,dy,3)$. A circle is specified by a center point and radius. A rectangle is specified by its lower-left and upper-right corners.

Suppose that our goal is to synthesize the program above given *just* the picture in fig. 3-1. As mentioned in section 3.1, a standard approach is to perform bottom-up search over programs in the DSL: i.e. create programs by composing together smaller terms but discard those that create previously seen shapes. This approach—known as bottom-up enumeration with equivalence reduction [104]—is a simple, powerful domain-agnostic synthesis algorithm that works in many domains.

However, the inverse-CSG domain is full of programs that are *similar* but not

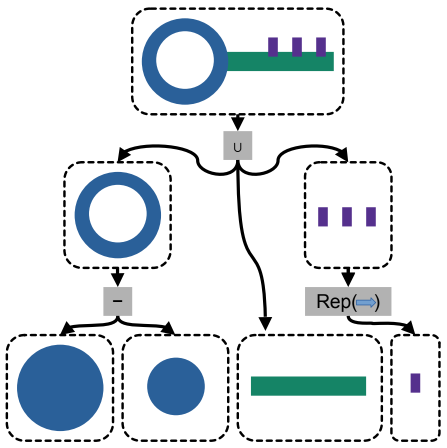


Figure 3-2: The solution, illustrated.

identical. To see why, consider the two circles that make up the handle of the key. If the outer circle was slightly larger or slightly shifted, it would still be clear to us that the image is only slightly perturbed. We would be able to fix the program by locally improving it—i.e., shifting the circle back into place by changing its parameters. We should not need to retain both programs in the search space, since one transforms straightforwardly into the other. However, we cannot use equivalence reduction to group these two programs together, even though our intuition tells us that they should be nearly interchangeable.

To synthesize the figure above, our algorithm proceeds in two phases. It first performs coarse-grained search to look for a program P that is *close to* matching the target image. Then it applies perturbations to P to find a repair that *exactly* matches the given image. We now explain these two phases in detail.

3.2.1 Global Coarse-grained Search

The first phase of our algorithm is based on bottom-up search and, like prior work [109], it builds a data structure that compactly represents a large space of programs. We represent the space of programs using a variant of a finite tree automaton (FTA) called an *approximate finite tree automaton* (XFTA) (section 3.3.3). The key idea is to group together values that are semantically similar: in the CSG context, images that are sufficiently similar to each other are represented using the same state in the automaton.

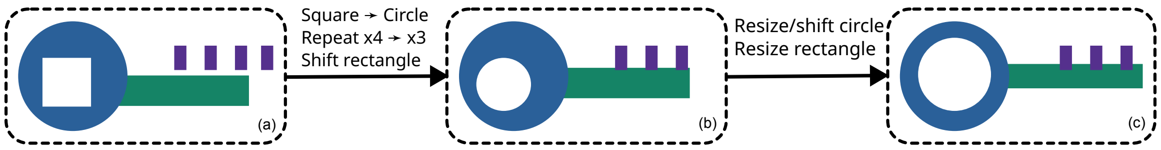


Figure 3-3: A sketch of the local-search process for the key example.

We construct this XFTA in three phases: *expansion*, *grouping*, and *ranking*. During expansion, operators are applied to subprograms to create new candidate programs. For our running example, the first expansion step generates the set of primitive shapes. Later expansions compose shapes together using set operators and the `Repeat` construct to create images of increasing complexity.

In the grouping phase, images are clustered. Each cluster has a center c and radius ϵ, and every image in the cluster is within distance ϵ of c. Although every image in the cluster is retained as part of the search space, only the center of the cluster participates in further expansion steps. The clustering phase is essentially a relaxed version of equivalence reduction.

Finally, in the ranking phase, the w clusters that are closest to the goal image are retained, with the goal of focusing search on the programs that are likely to produce the goal. After ranking, the top w clusters are inserted as new states into the XFTA, and the operators that produced each state in the cluster are inserted as edges.

When the global search terminates, the XFTA represents a space of programs that are close to the target image but it may not contain exactly the target image. To address this issue, our method performs a second level of *local search*.

3.2.2 Local Fine-grained Search

The local search proceeds in two phases: it extracts a candidate program from the XFTA, and it attempts to repair the candidate.

Since each node in the XFTA represents a (possibly) exponentially large set of programs, we use a greedy algorithm to select a program from this set rather than attempting to exhaustively search over it. Starting at an accepting state of the automaton, the program extractor selects the incoming edge that produces the image closest to the goal. Selecting an edge determines the root operator of the candidate program and the program sets from which to select arguments. Extraction proceeds recursively, always minimizing the distance between the overall candidate program and the goal.

When a candidate program has been extracted, we attempt to *repair* it by applying syntactic rewrites. The sequence of rewrites is chosen by a form of tabu search [42] and is guided by the distance from the candidate program to the goal. At each step of the repair process, we consider the set of programs obtained by applying a single rewrite rule to the current program and choose the closest to the goal. This process continues until the desired program is found or until a maximum number of rewrites have been applied.

Figure 3-3 gives a high-level view of this repair process. Starting from a program whose output is similar to the input image, the repair process applies rewrites such as changing squares to circles or incrementing/decrementing numeric parameters. Each rewrite gets us closer to the target image so the local search can often quickly converge to a program that exactly produces the target.

3.3 Metric Program Synthesis Algorithm

In this section, we describe our proposed synthesis algorithm. Given a language L and a set of input-output examples of the form $\{(I_1, O_1), \ldots, (I_n, O_n)\}$, our goal is to synthesize a program P in L such that $\forall i \in [1, n].\ [\![P]\!](I_i) = O_i$, meaning that evaluating P on I_i yields O_i for every example.

The rest of this section is organized as follows: First, because our method builds on bottom-up synthesis using finite tree automata, we start with preliminary information on FTAs in section 3.3.1. Then, in section 3.3.2, we describe our top-level synthesis algorithm, followed by discussions of its three subprocedures in section 3.3.3 through section 3.3.5.

3.3.1 Background on Synthesis using FTAs

Our synthesis algorithm builds on prior work on synthesis using finite tree automata (FTA). At a high level, an FTA is a generalization of a DFA from words to trees; just as a DFA accepts words, an FTA recognizes trees. FTAs are defined as follows:

Definition 1. (FTA) *A bottom-up finite tree automaton (FTA) over alphabet* Σ *is a tuple* $\mathcal{A} = (Q, Q_f, \Delta)$ *where* Q *is the set of states,* $Q_f \subseteq Q$ *are the final states, and* Δ *is a set of transitions of the form* $\ell(q_1, \ldots, q_n) \to q$ *where* $q, q_1, \ldots, q_n \in Q$ *and* $\ell \in \Sigma$*.*

FTAs are useful in synthesis because they can compactly encode sets of programs represented by their abstract syntax trees [109, 108, 69]. When used for synthesis, states of the FTA correspond to values (e.g., integers), and the alphabet corresponds to the set of DSL operators (e.g., $+, \times$). Final states are marked based on the specification, and transitions model the semantics of the underlying DSL. For instance, in a language with a negation operator $\neg$, transitions $\neg(0) \to 1$ and $\neg(1) \to 0$ express the semantics of negation.

We can view terms over an alphabet Σ as trees of the form $T = (n, V, E)$ where n is the root node, V is a set of labeled vertices, and E is the set of edges. A term T is said to be accepted by an FTA if T can be rewritten to some state $q \in Q_f$ using transitions Δ. Finally, the language of a tree automaton $\mathcal{A}$ is denoted as $\mathcal{L}(\mathcal{A})$ and consists of the set of all terms accepted by $\mathcal{A}$.

Given a specification φ, the idea behind FTA-based synthesis is to build an FTA whose language is the set of all programs satisfying φ. FTAs can be used to solve PBE problems as follows: For each input-output example (I, O), start with a state representing I and construct new states and transitions by applying the DSL operators. For example, given FTA states representing integers 1 and 2 and a $+$ operator in the DSL, we generate a new state representing 3 using the transition $+(1, 2) \to 3$. This process of adding new states and transitions to the FTA continues until either there are no more states to be added or a predefined bound on the number of states or transitions has been reached. The final state of the FTA corresponds to the output example O, and the standard intersection operator is used to generate an FTA whose language includes programs that are consistent with *all* input-output examples.

As this discussion makes clear, an FTA-based approach can compactly represent the version space if many DSL programs share the *same* input-output behavior (because such programs lead to the same FTA state). FTA-based synthesis may not scale in application domains that do not have this property. Prior work on FTA-based synthesis has tried to tackle this problem using abstract interpretation and abstraction refinement [109]: in that setting, FTA states correspond to *abstract* rather than *concrete* values, and transitions are constructed using the *abstract* semantics of the DSL. Since an *abstract FTA* overapproximates the set of programs consistent with the specification, one needs to perform abstraction refinement to iteratively rule out spurious programs from the language of the FTA. While this so-called SYNGAR approach has proven to be effective in some domains like tensor manipulations [109], such abstractions are not always easy to construct. For example, we have found the inverse CSG domain to *not* be amenable to an abstract-interpretation approach. Our metric-based synthesis algorithm is an attempt to solve this problem in a different way using distances rather than abstract domains.

3.3.2 Overview of Synthesis Algorithm

As mentioned earlier, the key idea behind metric-based synthesis is to relax the *observational equivalence* criterion into *observational similarity* by using a distance metric. More formally, we define observational similarity as follows:

Definition 2. (Similarity) *Two values v and v' are* similar*, denoted $v \simeq_\epsilon v'$ if they are within ϵ of each other, according to a distance metric δ:*

$$v \simeq_\epsilon v' \Leftrightarrow \delta(v, v') \leq \epsilon.$$

The main idea behind our synthesis algorithm is to construct an approximate version space by clustering together values that are similar. Just as the SYNGAR idea of [109] groups together values based on an *abstract* notion of observational equivalence, our method groups together values based on this notion of *similarity* and

Algorithm 1 Metric synthesis algorithm.

Require: Σ is a set of operators, I and O are the input and output examples respectively, c_{max} is the maximum program size to consider when constructing the XFTA, w is the beam width, δ is a distance metric between values, ϵ is the threshold for clustering.

Ensure: On success, returns a program p where $[\![p]\!](I) = O$. On failure, returns $\bot$.

```
1: procedure MetricSynth($\Sigma, I, O, c_{max}, w, \delta, \epsilon$)
2:     $\mathcal{A} \leftarrow$ ConstructXFTA($\Sigma, I, O, c_{max}, w, \delta, \epsilon$)
3:     for $P \in$ Extract($\mathcal{A}, I, O, q, \delta$) do
4:         $P \leftarrow$ Repair($I, O, \delta, P$)
5:         if $P \neq \bot$ then return $P$
6:     return $\bot$
```

constructs a so-called *approximate FTA (XFTA)* representing programs that produce values close to the desired output.

Our top-level metric program synthesis approach is presented in algorithm 1 and is parameterized over a (1) distance metric δ, (2) radius ϵ, and (3) domain-specific language L. At a high level, this algorithm consists of three steps:

1. **XFTA construction:** First, MetricSynth constructs an FTA that represents a space of programs that produce values close to the goal (line 2 of algorithm 1). However, because this FTA is constructed by grouping similar values together, a program accepted by this automaton does not necessarily satisfy the specification.

2. **Program extraction:** To deal with the approximation introduced by clustering, the algorithm enters a loop in which it repeatedly extracts programs from the FTA using the call to Extract at line 3. The goal of Extract is to find a program in the language of the FTA that produces a value that is sufficiently close to the target.

3. **Program repair:** Because the extracted program does not satisfy the input-output examples in the general case, the Repair procedure (invoked at line 4) tries to find a syntactic perturbation of P that exactly satisfies the input-output

Algorithm 2 Algorithm for constructing an approximate FTA.

Require: Σ is a set of operators, all other parameters are the same as in Algorithm 1. k is a hyper-parameter that determines the number of states that the automaton should accept.

Ensure: Returns an XFTA.

```
1: procedure ConstructXFTA($\Sigma, I, O, c_{max}, w, \delta, \epsilon$)
2:     $Q \leftarrow I, \Delta \leftarrow \emptyset$
3:     for $1 \leq c \leq c_{max}$ do
4:         $\Delta_{frontier} \leftarrow \{\ell(q_1, \ldots, q_n) \rightarrow q \mid \ell \in \Sigma, \{q_1, \ldots, q_n\} \subseteq Q, [\![\ell(q_1, \ldots, q_n)]\!] = q\}$
5:         $(Q_c, \Delta_c) \leftarrow$ Cluster($\Delta_{frontier}, \delta, \epsilon$)
6:         $Q' \leftarrow$ TopK($Q_c, \delta(O), w$)
7:         $Q \leftarrow Q \cup Q'$
8:         $\Delta \leftarrow \Delta \cup \{(\ell(q_1, \ldots, q_n) \rightarrow q) \mid q \in Q', (\ell(q_1, \ldots, q_n) \rightarrow q) \in \Delta_c\}$
9:     $Q_f \leftarrow$ TopK($Q, \delta(O), k$)
10:    return $(Q, Q_f, \Delta)$
```

examples. As we discuss in more detail in section 3.3.5, the repair procedure is based on rewrite rules and performs a form of tabu search, using the distance metric as a guiding heuristic.

We now discuss the ConstructXFTA, Extract, and Repair procedures in detail.

3.3.3 Approximate FTA Construction

Algorithm 2 shows our technique for constructing an approximate FTA for a given set of input-output examples. At a high level, this algorithm builds programs in a bottom-up fashion, clustering together those programs that produce similar values on the same input. To ensure that the algorithm terminates, it builds programs up to a fixed depth controlled by the hyperparameter c_{max}.

In more detail, ConstructXFTA adds new automaton states and transitions (initialized to I and $\emptyset$ respectively) in each iteration of the while loop. For each (n-ary)

Algorithm 3 Greedy algorithm for clustering states.

Require: Δ is a set of FTA transitions, all other parameters are the same as in Algorithm 1.

Ensure: Returns a set of FTA transitions.

```
1: procedure CLUSTER($\Delta, \delta, \epsilon$)
2:     $Q' \leftarrow \emptyset$, $\Delta' \leftarrow \emptyset$                ▷ New (clustered) states and transitions
3:     for $(\ell(q_1, \ldots, q_n) \to q) \in \Delta$ do
4:         $close \leftarrow \{q_{center} \in Q' \mid q_{center} \simeq_\epsilon q\}$
5:         if $close = \emptyset$ then $close \leftarrow \{q\}$
6:         $Q' \leftarrow Q' \cup close$,   $\Delta' \leftarrow \Delta' \cup \{\ell(q_1, \ldots, q_n) \to q' \mid q' \in close\}$
7:     return $(Q', \Delta')$
```

DSL operator ℓ and existing states $q_1, \ldots, q_n$, it gets a new *frontier* of candidate transitions $\Delta_{frontier}$ by evaluating $\ell(q_1, \ldots, q_n)$. The construction of this frontier corresponds to the *expansion phase* mentioned in section 3.2.

The expansion phase can produce many new states, making XFTA construction prohibitively expensive. Thus, in the next *clustering phase* (line 5 of algorithm 2), the algorithm groups similar states introduced by expansion into a single state (algorithm 3). Standard clustering algorithms like k-means are not suitable in this context because they fix the number of clusters but allow the radius of each cluster to be arbitrarily large. Instead, we would like to minimize the number of clusters while ensuring that the radius of each cluster is bounded. Hence, we use the CLUSTER procedure from algorithm 3 to generate a set of clusters where each state is within some ϵ distance from the center of a cluster. To do so, algorithm 3 iterates over the new states q in the frontier and starts a new cluster for q if none of the previous frontier states are within ϵ of q (lines 4–5 in algorithm 3). Otherwise, q is added to an existing cluster (line 6 in algorithm 3). For each new transition $\ell(q_1, \ldots, q_n) \to q$ of the frontier, clustering produces new transitions of the form $\ell(q_1, \ldots, q_n) \to q_c$ where q_c is the center of a cluster that q belongs to. Clustering produces a new set of states Q_c and a new set of transitions Δ_c to add to the automaton (line 5 of algorithm 2).

The clustering phase is greedy, and it is sensitive to the ordering of the transitions in Δ. Δ is sorted in our implementation, so the clusters are the same for every run of the algorithm. However, we do not experiment with multiple orders. This is a possible area of exploration for future work.

The final step in XFTA construction is the *ranking phase* (lines 6–8 of algorithm 2). Even after clustering, the automaton might end up with a prohibitively large number of new states, so ConstructXFTA only keeps the top w clusters in terms of their distance to the goal. Thus, in each iteration, algorithm 2 only ends up adding w new states to the automaton, like beam search.

3.3.4 Extracting Programs from XFTA

Algorithm 4 Algorithm for extracting programs from an XFTA.

```
Require: $\mathcal{A}$ is an XFTA; all other parameters are the same as in Algorithm 1.
Ensure: Yields program terms that are accepted by $\mathcal{A}$.
1: procedure Extract($\mathcal{A}, I, O, \delta$)
2:     $Q_f \leftarrow$ FinalStates($\mathcal{A}$)
3:     Sort $Q_f$ by $\delta(O)$ increasing.
4:     for $q_f \in Q_f$ do
5:         $\Delta_{root} \leftarrow \{(\ell(q_1, \ldots, q_n) \rightarrow q_f) \mid (\ell(q_1, \ldots, q_n) \rightarrow q_f) \in$ Transitions($\mathcal{A}$)$\}$
6:         yield ExtractTerm($\Delta_{root}, I, \delta(O)$)

7: procedure ExtractTerm($\Delta, I, \delta$)
8:     Let $(\ell(q_1, \ldots, q_n) \rightarrow q) \in \Delta$ be a transition where $q$ minimizes $\delta(q)$
9:     for $1 \leq i \leq n$ do                    ▷ Extract a program for each argument to $\ell$.
10:        $\delta_i \leftarrow \lambda q.\ \delta([\![\ell(q'_1, \ldots, q'_{i-1}, q, q_{i+1}, \ldots, q_n)]\!])$
11:        $p_i \leftarrow$ ExtractTerm($\Delta, I, \delta_i$)
12:        $q'_i \leftarrow [\![p_i]\!](I)$
13:    return $\ell(p_1, \ldots, p_n)$
```

We now turn our attention to the Extract procedure for picking a program that is accepted by our approximate FTA. Recall that programs accepted by the XFTA are not necessarily consistent with the input-output examples due to clustering.

Furthermore, two programs P, P' that are accepted by the XFTA need not be equally close to the goal state; for example, $[\![P]\!](I)$ might be much closer to O than $[\![P']\!](I)$ according to the distance metric δ. Ideally, we would like to find the best program that is accepted by the FTA (in terms of its proximity to the goal); however, this can be prohibitively expensive, as the automaton (potentially) represents an exponential space of programs. Thus, rather than finding the best program accepted by the automaton, our Extract procedure greedily selects a sequence of "good enough" programs in a computationally tractable way.

The high-level idea behind Extract is to recursively build a program starting from a final state q_f via the call to the recursive procedure ExtractTerm. At every step, the algorithm picks a transition $\ell(q_1, \ldots, q_n) \to q$ whose output minimizes the distance from the goal and then recursively constructs the arguments $p_1, \ldots, p_n$ of ℓ. Note that this algorithm is greedy in the sense that it tries to find a single operator that minimizes the distance from the goal rather than a sequence of operators (i.e., the whole program). Hence, there is no guarantee that Extract will return the optimal program accepted by $\mathcal{A}$.

3.3.5 Distance-Guided Program Repair

The final part of our synthesis algorithm (Repair) takes the program that was extracted from the XFTA and attempts to repair it by applying syntactic rewrite rules. In particular, given a program P that is close to the goal, Repair tries to find a program P' that is (1) syntactically close to P and (2) correct with respect to the input-output examples (i.e., $[\![P]\!](I) = O$).

Our Repair procedure is parameterized by a set of rewrite rules R of the form $t \to s$. We say that a program P can be rewritten into P' if there is a rule $r = (t \to s) \in R$ and a substitution σ such that $P = \sigma t$ and $P' = \sigma s$. We denote the application of rewrite rule r to P as $P \to_r P'$.

The Repair procedure is presented in algorithm 5 and applies goal-directed rewriting to the candidate program, using the distance function δ to guide the search. In particular, it starts with the input program P and iteratively applies a rewrite rule until either a correct program is found or a bound n on the number of rewrite rules is reached. In each iteration of the loop (lines 3–6), it first generates a set of new candidate programs (called *neighbors*) by applying a rewrite rule to P and (greedily) picks the program P' that minimizes the distance $\delta(O, [\![P']\!](I))$. In the next iteration, the new program P' is used as the seed for applying rewrite rules.

Algorithm 5 Algorithm for repairing a program.

Require: I, O are the input-output examples, δ is a distance metric, P is a program. There are also two hyperparameters: n is the maximum number of rewrites to perform, and R is a set of rewriting rules.

Ensure: Returns a program P' such that $[\![P']\!](I) = O$ or returns $\perp$.

```
1: procedure Repair($I, O, \delta, P$)
2:     $S \leftarrow \emptyset$
3:     while $i < n$ do
4:         $neighbors \leftarrow \{P' \mid P \rightarrow_r P', r \in R\} - S$
5:         $P \leftarrow \arg\min_{p \in neighbors} \delta(O, [\![p]\!](i))$
6:         if $[\![P]\!](I) = O$ then return $P$
7:         $S \leftarrow S \cup \{P\}$
8:     return $\perp$
```

Note that our Repair procedure uses a (bounded) set S to avoid getting stuck in local minima, as in tabu search [42]. The set S contains the most recently explored k programs, and, when applying a rewrite rule, the Repair procedure avoids generating any program in S.

3.3.6 Theoretical Guarantees

As stated earlier, our synthesis algorithm does not come with completeness guarantees *in general*, but it does so under certain assumptions. While these conditions are fairly strong and not likely to be met in many application domains of interest, we believe stating these conditions is useful for successfully instantiating metric synthesis in new domains.

First, we give notation for local search (section 3.3.5) reachability. Let $P \rightsquigarrow P'$

denote that $\exists P''.\ [\![P']\!] = [\![P'']\!] \wedge \textsc{Reach}(P, P'')$, which says that a program equivalent to P' is reachable from P via local search. We say that $P \leftrightsquigarrow P'$ iff $P \rightsquigarrow P' \wedge P' \rightsquigarrow P$. We note that hill-climbing search is generally symmetric (and is for our domains); if the local search is symmetric, then $P \rightsquigarrow P' \vee P' \rightsquigarrow P \implies P \leftrightsquigarrow P'$.

Assumption #1. First, for our theoretical analysis, we assume that the distance function and local search are related. We formalize this relationship using the notion of *local reachability*:

Definition 3. ***(Local reachability)*** *The* local reachability *property requires:*

$$[\![P]\!] \simeq_\alpha [\![P']\!] \implies P \leftrightsquigarrow P'$$

Intuitively, this property states that programs that are within some small distance α from each other can be rewritten to one another through a small number of applications of the rewrite rules. Hence, the local reachability property depends on the chosen set of rewrite rules for the target application domain.

Assumption #2. Our second assumption is the so-called *directionality* property, which, intuitively, states that subprograms of the target program should *not* be prohibitively far away from the goal:

Definition 4. ***(Directionality)*** *Let $P^\star$ be the desired solution for the synthesis task. The monotonicity assumption requires:*

$$\forall P, C.\ \delta([\![P]\!], [\![P^\star]\!]) > \beta \implies C[P] \not\leftrightsquigarrow P^\star.$$

This definition says that if a program P is far enough from the goal $P^\star$, then it is not a useful subprogram; there is no context C that we can place it in that is in the neighborhood of $P^\star$. If this property does not hold, then our algorithm may prune useful subprograms from the search space.

Assumption #3. Finally, we assume that all DSL operators preserve the ability to find programs with local search. We refer to this property as *transparency*:

Definition 5. ***(Transparency)*** *The transparency property requires that, for every n-ary DSL operator f, we have:*

$$\bigwedge_{i=1}^{n} a_i \leftrightsquigarrow b_i \implies f(a_1, \ldots, a_n) \leftrightsquigarrow f(b_1, \ldots, b_n)$$

Along with local reachability, this assumption justifies the use of grouping, because it says that retaining the center of a group suffices to recover the rest of the programs in the group.

Under these assumptions, we can state the following completeness property of metric synthesis:

Theorem 2. *Let $P^\star$ be a program in a language L, consistent with I/O examples (I, O), where $|P^\star| < c_{max}$, and where the beam width w is large enough that all programs within β of $P^\star$ are retained. Under the three assumptions discussed above,* $\textsc{MetricSynth}(L, I, O, c_{max}, w, \delta, \alpha/2)$ *will return a program equivalent to $P^\star$.*

To prove the completeness theorem, we first prove the following lemma about the XFTA produced by ConstructXFTA. We denote "a state q accepts a program P" as $P \in q$. We say that for $\mathcal{A} = (Q, Q_f, \Delta)$, a program equivalent to $P^\star$ is reachable from $\mathcal{A}$ iff $\exists q \in Q_f . \forall P \in q.\ P \leftrightsquigarrow P^\star$, and we denote this as $\mathcal{A} \overset{*}{\rightsquigarrow} P^\star$.

Lemma 1. $\textsc{ConstructXFTA}(\Sigma, I, O, c_{max}, w, \delta, \alpha) = \mathcal{A}$ *and for all subterms P of $P^\star$, $\mathcal{A} \overset{*}{\rightsquigarrow} P$ (under the same assumptions as in theorem 2).*

Proof. The proof is by induction on the size c of the largest programs in $\mathcal{A}_i$, where $\mathcal{A}_i$ is produced by the ith iteration of the loop in ConstructXFTA.

Base case. Let P be a subterm of $P^\star$ s.t. $|P| = 1$. We show that $\mathcal{A}_1 \overset{*}{\rightsquigarrow} P$. After building $\Delta_{frontier}$, we discard transitions $\ell \to q$ that are not in top-k. For a suitably chosen w, the top-k transitions are within distance β from the goal, so we have $\delta(q, [\![P^\star]\!]) > \beta$. By directionality, the transition labeled by P is retained because there is a context C s.t. $C[P] = P^\star$. We then cluster the transitions into groups of radius at most $\alpha/2$. By local reachability, P is reachable from any of the programs in its group. Therefore, $\mathcal{A}_1 \overset{*}{\rightsquigarrow} P$.

Inductive case. Let $P = a \oplus b$ be a subterm of $P^\star$ s.t. $|P| = c$. We assume a binary operator for simplicity of presentation. By induction, $\mathcal{A}_{c-1} \overset{*}{\rightsquigarrow} a$ and $\mathcal{A}_{c-1} \overset{*}{\rightsquigarrow} b$. Let q_a and q_b be the representative states of a and b.

We now show that the transition labeled by $\oplus(q_a, q_b)$ is retained after ranking. Let $a' \in q_a$ and $b' \in q_b$. By induction, $a' \leftrightsquigarrow a$ and $b' \leftrightsquigarrow b$. P is a subterm of $P^\star$, so there is some C s.t. $C[P] = C[a \oplus b] = P^\star$. By transparency, $C[a' \oplus b'] \leftrightsquigarrow C[a \oplus b]$, so by directionality, $\delta(\llbracket P \rrbracket, \llbracket P^\star \rrbracket) \leq \beta$, which means that the transition is retained. After clustering, P is reachable from any program in its group, by local reachability. Therefore, $\mathcal{A}_c \overset{*}{\rightsquigarrow} P$. $\square$

Observe that the completeness theorem follows immediately from this lemma: Let $\mathcal{A}$ be the result of ConstructXFTA$(\Sigma, I, O, c_{max}, w, \delta, \alpha/2)$. For each final state q in $\mathcal{A}$, we extract a program $P \in q$ and perform local search. By the above lemma, there must be some q such that for any P, $P \leftrightsquigarrow P^\star$. Therefore, we always find a program equivalent to $P^\star$.

3.4 Instantiating Metric Synthesis in Application Domains

The MetricSynth algorithm can be instantiated in different application domains by supplying a suitable DSL, distance metric, and rewrite rules for repair. In this section, we discuss how to instantiate it in the inverse CSG, regular-expression synthesis, and tower-building domains.

3.4.1 Instantiation for Inverse CSG

The *inverse CSG problem* aims to "decompile" a complex geometric shape into a set of geometric operations that were used to construct it. We now describe how we use

$$M = \{(u,v) \mid 0 \leq u < x_{max}, 0 \leq v < y_{max}\}$$

$$\textsc{Eval}(\texttt{Circle}(x,y,r)) = \left\{(u,v) \mid \sqrt{(x-u)^2 + (y-v)^2} < r, (u,v) \in M\right\}$$

$$\textsc{Eval}(\texttt{Rect}(x_1,y_1,x_2,y_2)) = \left\{(u,v) \;\middle|\; \begin{array}{l} x_1 \leq u \wedge y_1 \leq v \wedge u \leq x_2 \wedge v \leq y_2, \\ (u,v) \in M \end{array}\right\}$$

$$\textsc{Eval}(\texttt{Repeat}(e,x,y,c)) = \left\{(u,v) \;\middle|\; \begin{array}{l} (u-ix, v-iy) \in \textsc{Eval}(e), \\ 0 \leq i < c, \\ (u,v) \in M \end{array}\right\}$$

$$\textsc{Eval}(e \cup e') = \textsc{Eval}(e) \cup \textsc{Eval}(e') \qquad \textsc{Eval}(e - e') = \textsc{Eval}(e) \setminus \textsc{Eval}(e')$$

Figure 3-4: The semantics of CSG programs, given as an evaluation function.

our metric-program-synthesis framework to solve the inverse CSG problem.

Domain-Specific Language

The syntax of our inverse CSG DSL is:

$$E \;:=\; \texttt{Circle}(x,y,r) \mid \texttt{Rect}(x_1,y_1,x_2,y_2) \mid E \cup E \mid E - E \mid \texttt{Repeat}(E,x,y,c).$$

This DSL includes two primitive shapes: circles and rectangles. A circle is represented by a center coordinate and a radius. A rectangle is axis-aligned and is represented by the coordinates of its lower-left and upper-right corners. The primitive shapes can be combined using union, difference, and repeat operators. $\texttt{Repeat}(E,x,y,c)$ takes an image E, a translation vector $v = (x,y)$, and a count c, and it produces the union of E repeated c times, translated by v. For example: $\texttt{Repeat}(\texttt{Circle}(v,r),v',2) = \texttt{Circle}(v,r) \cup \texttt{Circle}(v+v',r)$. $\texttt{Repeat}$ allows programs with repeating patterns to be expressed compactly, which also makes these programs easier to synthesize.

Figure 3-4 presents the semantics of our CSG DSL using an $\textsc{Eval}$ procedure; $\textsc{Eval}$ takes a program and produces a bitmap image. Bitmap images are represented as the set of all filled pixels (u,v).

Synthesis Problem

Given a bitmap image B of size $w \times h$, inverse CSG aims to synthesize a program P such that:

$$\forall 0 \leq x < w, 0 \leq y < h.\ B(x, y) \iff (x, y) \in [\![P]\!]$$

Note that inverse CSG is exactly a programming-by-example (PBE) problem: We can think of the bitmap image as a set of I/O examples where each input example is a pixel (x, y) and the output example is a Boolean. However, a key difference from standard PBE is that the number of examples we need to deal with is quite large: for instance, in our evaluation, we use 32×32 bitmap images, so there are 1024 I/O examples.

Distance Function

The distance metric is a key component of our algorithm. For inverse CSG, we use a slight modification of the Jaccard distance that takes into account the goal value:

$$\delta_O(q, q') = 1 - \frac{|f_O(q) \cap f_O(q')|}{|f_O(q) \cup f_O(q')|} \quad \text{where } f_O(q) = \{(x, y, b) \mid (x, y) : b \in q, b \neq O[x, y]\}.$$

Intuitively, δ_O only considers the pixels that *differ* from the goal image O. This is desirable because it amplifies small differences between images that are close to the goal, as illustrated in fig. 3-5. The original Jaccard distance does not satisfy local reachability (definition 3), because there are programs whose output is largely similar, but differ in many small details that require significant transformation to add. We started with the Jaccard distance and changed it in response to the observation that the search was clustering programs that were not locally reachable (see fig. 3-14). While this distance also does not satisfy local reachability in general, it gets closer as the search progresses and the space fills with programs that are similar to the output.

Theorem 3. *δ_O is a metric on the set of images.*

Proof. Let O be some goal value. Let $\delta_J(q, q') = 1 - \frac{|q \cap q'|}{|q \cup q'|}$ be the Jaccard distance, which is a metric on finite sets. We have $\delta_O(q, q') = \delta_J(f_O(q), f_O(q'))$ where f_O (defined

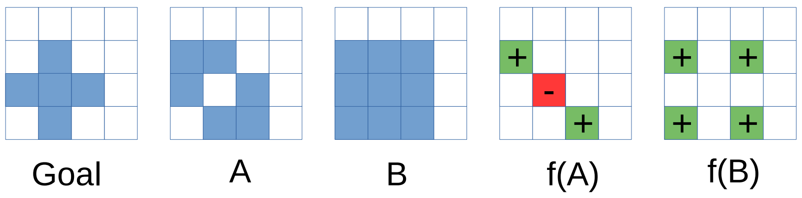


Figure 3-5: Illustration of the distance transformation f_O. Note that the Jaccard distance is $\delta_J(A, B) = \frac{1}{3}$, whereas $\delta_{Goal}(A, B) = \frac{3}{5}$. This shows how the distance between images that are very close to the goal is magnified.

above) is a function $Image \to \mathbb{Z} \times \mathbb{Z} \times \mathbb{B}$. Because an injective function f from any set S to a metric space (M, δ) gives a metric $\delta(f(x), f(x'))$ on S, we can show that δ_O is a metric by showing that f is injective. To see why this is the case, note that we can define the inverse of f_O as follows:

$$f_O^{-1}(s) = \{(x, y) : b' \mid (x, y) : b \in O, b' = \neg b \text{ if } (x, y, \neg b) \in s \text{ else } b\}\,.$$

Intuitively, where f_O returns the differences between q and O, f_O^{-1} applies those differences to O to obtain q. $\square$

Rewrite Rules

Recall that our Repair procedure is parameterized over a set of rewriting rules R. The rules we use for inverse CSG modify integers (within bounds) and transform squares into circles (and vice-versa):

$$x \to x + 1 \qquad \text{if } x \text{ is an integer and } x < x_{max}$$

$$x \to x - 1 \qquad \text{if } x \text{ is an integer and } x > 0$$

$$\texttt{Circle}(x, y, r) \to \texttt{Rect}(x - r, y - r, x + r, y + r)$$

$$\texttt{Rect}(x_1, y_1, x_2, y_2) \to \texttt{Circle}(x_1 + r, y_1 + r, r) \qquad \text{if } r = \frac{x_2 - x_1}{2} \text{ and } x_2 - x_1 = y_2 - y_1$$

The circle-to-square rule in particular helps align the local search with the distance function in accordance with local reachability. Similarly-sized and positioned circles

Algorithm 6 Algorithm for checking that a CSG program is well-formed.

$$
\begin{aligned}
\mathrm{WF}(\texttt{Circle}(x, y, r)) &= r > 0 \land 0 \le x - r \land x + r < x_{max} \land 0 \le y - r \land y + r < y_{max} \\
\mathrm{WF}(\texttt{Rect}(x_1, y_1, x_2, y_2)) &= x < x' \land y < y' \\
\mathrm{WF}(e - e') &= \mathrm{WF}(e \cup e') = \mathrm{WF}(e) \land \mathrm{WF}(e') \\
\mathrm{WF}(\texttt{Repeat}(e, v, c)) &= \|v\| > 0 \land c > 1 \land \mathrm{WF}(e)
\end{aligned}
$$

and squares are similar according to δ, so we ensure they are close according to the local search.

Symmetry Breaking

The CSG DSL contains a significant number of *symmetric* programs: i.e. programs that are observationally equivalent but syntactically distinct. Although the existence of symmetric programs does not make the generation of the XFTA more expensive, it does make searching the resulting graph more difficult because it increases the number of paths that reach each state.

To mitigate this issue, we define a *canonical form* for CSG programs and modify the evaluation function to return $\perp$ for non-canonical programs. Furthermore, programs which evaluate to $\perp$ are not included in the XFTA.

The canonical form does not admit empty primitive shapes or primitive shapes that overlap the edge of the canvas. `Repeat` operators must move the shape and perform more than one repetition. The arguments to commutative operators like union must be ordered.

3.4.2 Instantiation for Regular Expressions

Our second application domain of metric program synthesis is generating regular expressions from a set of positive and negative examples.

Domain-Specific Language

Our regular-expression language is a subset of the regular-expression language used in prior work [15]. The syntax is:

$$\begin{aligned} E := {} & \emptyset \mid C \mid \texttt{Concat}(E, E) \mid \texttt{Repeat}(E, x) \\ & \mid \texttt{RepeatRange}(E, x_1, x_2) \mid \texttt{RepeatAtLeast}(E, x) \\ & \mid \texttt{Optional}(E) \mid E \wedge E \mid E \vee E \mid \neg E \end{aligned}$$

The DSL includes character classes, concatenation, repetition, optional matches, conjunction, disjunction, and negation. Character classes match a single character from a set; we use a set of single-character classes that includes all the printable characters as well as multicharacter classes for numbers, capital and lowercase letters, symbols, and vowels. $\texttt{Concat}(E, E')$ matches E followed by E'. For example, $\texttt{Concat}(\texttt{<num>}, \texttt{<a>})$ matches the string "0a." $\texttt{Repeat}(E, x)$ matches x repetitions of E. Programs in this DSL evaluate to a set of match locations in a string s. Formal semantics are given in Figure 3-6.

Synthesis Problem

As in prior work [62, 15], we consider the problem of synthesizing regular expressions from a given set of positive and negative examples. Let S^+ be a set of positive examples (strings), and let S^- be a set of negative examples. Then, the synthesis problem is to generate a regular expression E such that:

$$\forall s \in S^+.\ (0, |s|) \in \textsc{Eval}(E, s) \text{ and } \forall s \in S^-.\ (0, |s|) \notin \textsc{Eval}(E, s)$$

However, prior work has shown that synthesizing the *intended* regular expression just from positive and negative examples can be challenging if one only has access to a few examples. For this reason, [15] has advocated using *sketches* obtained from natural-language descriptions. Following that work, we consider a modified version of this problem where the regular expression needs to be a completion of the provided

$$
\begin{aligned}
\textsc{Eval}(C, s) &= \{(i, i+1) \mid 0 \le i \le |s|, s[i] \in C\} \\
\textsc{Eval}(\emptyset, s) &= \{(i, i) \mid 0 \le i \le |s|\} \\
\textsc{Eval}(\texttt{Concat}(E, E'), s) &= \left\{ (i, k) \;\middle|\; \begin{array}{l} (i, j) \in \textsc{Eval}(E, s), \\ (j', k) \in \textsc{Eval}(E', s), \\ j = j' \end{array} \right\} \\
\textsc{Eval}(\texttt{Repeat}(E, 1), s) &= \textsc{Eval}(E, s) \\
\textsc{Eval}(\texttt{Repeat}(E, x), s) &= \textsc{Eval}(\texttt{Concat}(E, \texttt{Repeat}(E, x-1)), s) \\
\textsc{Eval}(\texttt{RepeatRange}(E, x_1, x_2), s) &= \bigcup_{x_1 \le i \le x_2} \textsc{Eval}(\texttt{Repeat}(E, i), s) \\
\textsc{Eval}(\texttt{RepeatAtLeast}(E, x), s) &= \textsc{Eval}(\texttt{RepeatRange}(E, x, |s|), s) \\
\textsc{Eval}(\texttt{Optional}(E), s) &= \textsc{Eval}(E \vee \emptyset, s) \\
\textsc{Eval}(E \wedge E', s) &= \textsc{Eval}(E, s) \cap \textsc{Eval}(E', s) \\
\textsc{Eval}(E \vee E', s) &= \textsc{Eval}(E, s) \cup \textsc{Eval}(E', s) \\
\textsc{Eval}(\neg E, s) &= \{(i, j) \mid 0 \le i \le j \le |s|, (i, j) \notin \textsc{Eval}(E, s)\}
\end{aligned}
$$

Figure 3-6: Semantics of the regular-expression DSL.

sketch *and* satisfy the examples. Specifically, our problem formulation requires a sketch given in the form of a program with holes, where the holes may contain constraints on the terms that must be used to fill the hole, as in [15].

Distance Function

We use a distance function that simply counts the examples that the regular expression matches. This is only weakly predictive of the value of a subprogram to the overall solution, but it does predict useful subprograms when the overall program is a disjunction.

Rewrite Rules

Some of the constructs used in the regex DSL take integers as arguments. As in the inverse CSG instantiation, we include rewrite rules that transform integers. We also include a rule that rewrites `Repeat` to `RepeatRange`, because δ treats some programs that use `Repeat` as similar to programs that use `RepeatRange`, so the local search needs to reflect that property.

$$
\begin{aligned}
x &\to x+1 && \text{if } x \text{ is an integer and } x < x_{max} \\
x &\to x-1 && \text{if } x \text{ is an integer and } x > 0 \\
\texttt{Repeat}(E,x) &\to \texttt{RepeatRange}(E,x,x)
\end{aligned}
$$

Sketch Constraints

The regular-expression synthesis problem is based on sketches, so our synthesizer must produce programs that satisfy the examples *and* match the sketch. Sketches consist of terms with holes of the form $?\{E_1, \ldots, E_n\}$. A term matches a hole if it contains a match for one of the sub-sketches in the hole. For example, `<num>` and ¬`<num>` both match ?{`<num>`}. We extend the DSL semantics to return the set of matching sketches and we assert that solutions match the input sketch. Terms with distinct match sets are infinitely far apart, so are never clustered.

3.4.3 Instantiation for Tower Building

Our third application domain is the tower-building task from prior work [33, 75] that is inspired by AI planning tasks. Given a set of blocks and target "tower" (i.e. configuration of these blocks), this task aims to generate the desired tower by (programmatically) controlling a robot arm.

Domain-Specific Language

The syntax of the tower-building DSL is:

$$E := \texttt{DropH} \mid \texttt{DropV} \mid \texttt{MoveBefore}(E, x) \mid \texttt{MoveAfter}(E, x) \mid E; E \mid \texttt{Loop}(x, E)$$
$$\mid \texttt{Embed}(E).$$

Programs in this language control a robot arm which can move left and right along a horizontal track and can drop horizontal or vertical blocks. The state of the program includes the x-position of the arm and the list of dropped blocks. The DSL includes operators for dropping blocks, moving the robot arm, sequencing, and looping. `DropH` and `DropV` both add a new (horizontal or vertical) block to the tower. The block will be placed on the highest block that is below the arm. The move operators both update the position of the arm; `MoveBefore` moves the arm and then executes E, while `MoveAfter` moves the arm after evaluating E. `Embed` gives the language a degree of modularity. It executes E and then resets the arm to wherever it was before. `Loop` repeats the body x times. Note that these semantics correspond to an idealized model where blocks fall in a straight line until they land on top of another block and blocks cannot topple.

Formal semantics are given in Figure 3-7. In these semantics, h is the horizontal position of the hand and bs is the list of blocks dropped. Each block is represented by a triple (x, y, k) where x and y are the horizontal and vertical positions and k is the kind of block—either horizontal or vertical.

$$\begin{aligned}
\textsc{Eval}(\texttt{DropH}, (h, bs)) &= (h, (h, \max_{h \le x < h+3} \textsc{Top}(x, bs), \texttt{H}) : bs) \\
\textsc{Eval}(\texttt{DropV}, (h, bs)) &= (h, (h, \textsc{Top}(h, bs), \texttt{V}) : bs) \\
\textsc{Eval}(\texttt{MoveBefore}(E, x), (h, bs)) &= \textsc{Eval}(E, (h + x, bs)) \\
\textsc{Eval}(\texttt{MoveAfter}(E, x), s) &= (h + x, bs) \text{ where } (h, bs) = \textsc{Eval}(E, s) \\
\textsc{Eval}(E; E', s) &= \textsc{Eval}(E', \textsc{Eval}(E, s)) \\
\textsc{Eval}(\texttt{Loop}(1, E), s) &= \textsc{Eval}(E, s) \\
\textsc{Eval}(\texttt{Loop}(x, E), s) &= \textsc{Eval}(\texttt{Loop}(x - 1, E), \textsc{Eval}(E, s)) \\
\textsc{Eval}(\texttt{Embed}(E), (h, bs)) &= (h, bs') \text{ where } (h', bs') = \textsc{Eval}(E, s) \\
\textsc{Top}(bs, x) &= \max\{y \mid (b, x', y) \in bs, x = x'\}
\end{aligned}$$

Figure 3-7: Semantics of the tower-building DSL.

Synthesis Problem

The input to the synthesizer is a set of blocks B. Each block is represented as a tuple (b, x, y) where $b \in \{\texttt{H}, \texttt{V}\}$ is the type of block (either horizontal 1×3 or vertical 3×1) and (x, y) is the block's position. B must be a valid tower, which means that the blocks must not overlap. The synthesis problem is to produce a program P such that $\textsc{Eval}(P, s_0) = B$, where $s_0 = (0, [\,])$ is the initial state with no blocks placed and the hand at $x = 0$.

Distance Function

The distance function for the tower-building domain is based on the insight that translating a tower along the x-axis is straightforward, so we want our distance function to be *translation-invariant*. Hence, two programs that build the same tower in nearby places should be deemed similar. Based on this intuition, we use the Jaccard distance to compare two towers, and we normalize them before we compare them so that their leftmost block is at $x = 0$:

$$\delta(s, s') = \delta_J(z(s), z(s')) \textit{ where } z((h, bs)) = (h, \{(b, x - x_{min}, y) \mid (b, x, y) \in bs\})$$

where δ_J is the Jaccard distance and z is the normalizing function.

The distance function for the tower-building domain embodies two key insights:

- Translating a tower along the x-axis is straightforward, so it is better to construct the right tower in the wrong place than the converse.
- The behavior of a tower program depends on its initial state. We can approximate a tower program by running it on some number of initial states.

Rewrite Rules

As in the other domains, some constructs in the tower-building DSL take integer-valued arguments. Hence, we use rewrite rules that allow incrementing and decrementing these integers, as in the inverse-CSG and regular-expression domains. These rules suffice to change loop-iteration counts and to modify the movement operators.

3.5 Implementation

We have implemented our synthesis technique in a new tool called SyMetric written in OCaml. In what follows, we describe some optimizations.

Randomization Our implementation of Repair considers a random subset of the rewrites when generating candidate programs to select from. This randomization compensates for the greedy nature of this algorithm by introducing the possibility of taking a locally suboptimal step that turns out to be globally optimal.

Incremental clustering Since the clustering technique is a significant cost of approximate FTA construction, our implementation performs a few modifications. In particular, instead of computing all clusters and then sorting them, it first sorts the transitions and uses the first k clusters that it finds. Furthermore, because the number of transitions in $\Delta_{\mathit{frontier}}$ can be very large in the ConstructXFTA algorithm, our implementation incrementally collects the top states in batches. This involves evaluating the frontier multiple times, rather than storing it, but we find that, in practice, we need only a small prefix of the sorted frontier. Finally, our implementation of the Cluster procedure uses an M-tree data structure [20] to facilitate efficient insertion and range queries.

Our instantiation of SyMetric also performs a few domain-specific optimizations for the inverse-CSG and regular-expression domains.

Optimizations for inverse CSG Our instantiation of SyMetric in the inverse-CSG domain incorporates three low-level optimizations. First, it represents images as packed bitvectors to reduce their sizes. Second, our evaluation function for the CSG DSL is memoized. Third, our implementation uses optimized, vectorized ISPC [80] implementations for bitvector operations, distance functions, and for CSG operators such as `Repeat`.

Optimizations for regular expressions. In our implementation of the regular-expression domain, we view the match sets as graphs where the positions in the string are the nodes and the matches are the edges. We represent these graphs as adjacency matrices using an efficient packed Boolean representation. This representation is particularly effective for synthesizing regular expressions from examples, because the example strings tend to be short, which mitigates the $O(n^2)$ memory cost of the matrix.

We also note that `Repeat`(E, n) matches (i, j) if E matches $(i, k_1), (k_1, k_2), \ldots, (k_{n-1}, j)$. That is, `Repeat`$(E, n)$ matches (i, j) if there is a walk of length n in the match graph for E from i to j. The walks of length n are given by the nth power of the adjacency matrix A_E^n. Therefore, we can build efficient implementations for the `Repeat*` operators and for `Concat` using efficient Boolean matrix multiplication and exponentiation. We implement these operations using the packed-bitvector library developed for the

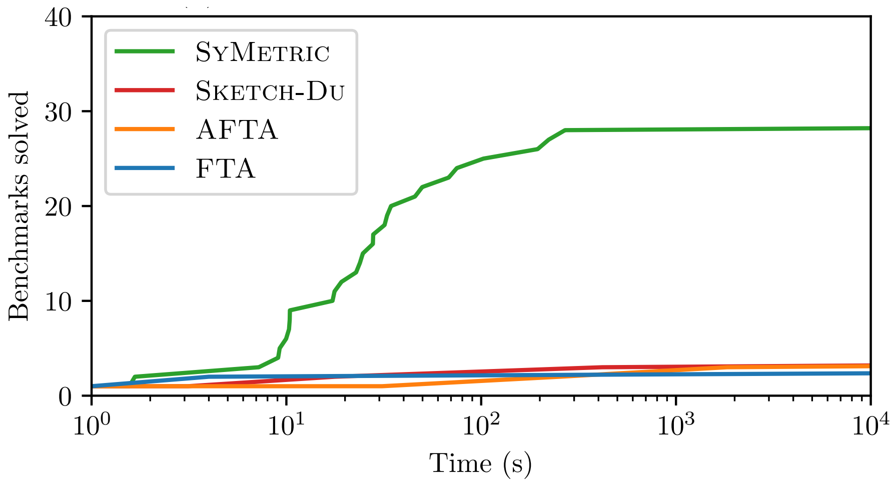


Figure 3-8: Synthesis performance on the inverse-CSG domain.

inverse CSG domain, and we use ISPC for vectorization.

3.6 Evaluation

In this section, we describe a series of experiments to empirically evaluate our approach. In particular, our experiments are designed to evaluate the following key research questions:

- **RQ1:** How does SyMetric compare against other domain-agnostic and domain-specific synthesis tools in the inverse-CSG, regular-expression, and tower-building domains?
- **RQ2:** What is the relative importance of the various ideas comprising our approach?
- **RQ3:** How do different components of our synthesis algorithm contribute to running time?

3.6.1 Inverse CSG

In our evaluation on the inverse CSG domain, we compare SyMetric against the following baselines:

- Sketch-Du: Since prior work on inverse CSG is based on the Sketch synthesis system, we implement a baseline that uses an encoding like the one used in InverseCSG [31]. However, since that work focuses on 3D shapes, we modify the encoding to work with our DSL for 2D geometry and also add support for repetition (see section 3.7). Unlike the encoding in [31], our sketches do not contain hints about the locations of primitive shapes, so the synthesizer needs to discover the numeric parameters of these primitive shapes.

- FTA: This baseline performs bottom-up synthesis (with equivalence reduction) using FTAs [108]. Like the implementation of SyMetric, this baseline is also implemented in OCaml. Note that this baseline only adds FTA states and transitions until a final state is reached, as our goal is to find *one*, rather than all, programs consistent with the specification.

- AFTA: This baseline performs bottom up synthesis with abstract FTAs (AFTAs) [109]. In particular, this method uses abstract values as states of the FTA, constructs FTA transitions using the abstract semantics, and performs abstraction refinement to deal with spurious programs extracted from the AFTA. Our implementation of this baseline is also in OCaml and uses the matrix abstract domain from [109] since bitmap images can be viewed as matrices.

Benchmarks Since prior work on Inverse CSG [31, 56] mostly targets 3D benchmarks, we construct our own benchmark suite for 2D Inverse CSG. We consider a total of 40 benchmarks, where 25 correspond to outputs of randomly generated programs (modulo some nontriviality constraints) and 15 are handwritten benchmarks of visual interest.

Setup We run these experiments on a machine with two AMD EPYC 7302 processors (64 threads total) and 256 GB of RAM. We use a time limit of 1 hour and a memory limit of 4 GB (so that we can run many benchmarks at the same time). For the hyperparameters for SyMetric, we use $\epsilon = 0.2$, $w = 200$, and the maximum number of rewriting steps is $n = 500$.

Summary of results The results of this evaluation are shown in fig. 3-8. SyMetric can solve 70% of these benchmarks, but the baselines fail on all of them except at most 3. In what follows, we discuss why the baselines perform poorly and the failure cases for SyMetric.

FTA results The FTA baseline fails on all but the smallest of the handwritten benchmarks. When it fails, it is universally because it runs out of memory. For this domain, few programs produce exactly the same output image, so equivalence reduction is not enough to reduce memory consumption.

AFTA results We found that the AFTA baseline cannot solve *any* benchmarks when we include the `Repeat` operator in the DSL, as repetition causes difficulties with Syngar's abstraction-refinement phase, specifically because refining one pixel can cause the refinement of multiple other pixels, which causes the abstraction to track an increasing number of pixels over multiple refinement iterations [109]. However, the AFTA baseline can solve 3 of the 40 benchmarks if we omit the `Repeat` operator from the DSL. For many of the remaining benchmarks, the target programs become quite complex without the `Repeat` operator, so the AFTA approach fails either because it reaches the time limit or fails to find a program with the AST depth limit of 40.

Sketch-Du results Our third baseline Sketch-Du—an adaptation of InverseCSG [31] to our setting—can solve three of the handwritten benchmarks but fails on the remaining ones. For the 37 benchmarks it cannot solve, it runs out of memory 75% of the time and runs out of time the remaining 18%. We attempted to provide this baseline with parameters that would minimize its memory use and maximize its chances of successfully completing the benchmarks. To that end, we used Sketch's specialized integer solver to reduce memory overhead; we controlled the amount of unrolling in the sketch based on the size of the benchmark program, and we used Sketch's example-file feature, which reduces the time to find counterexamples during the CEGIS loop. However, even with these optimizations, we found that the large number of examples in the inverse-CSG domain (one per pixel, so 1024 total) causes

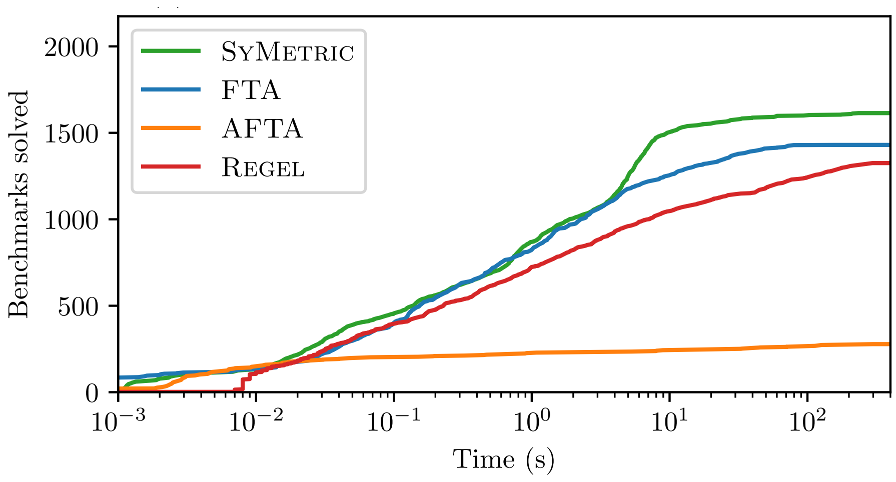


Figure 3-9: Synthesis performance on the regular-expression domain.

Sketch to perform many iterations of the CEGIS loop.

SyMetric results SyMetric performs significantly better than the other baselines, solving 70% of the benchmarks. Based on our manual inspection, we found two dominant failure modes. One of them is that the beam width w may be too narrow, causing critical subprograms to be dropped from the search space. This effect is more pronounced when the benchmark relies on subprograms that are far from the goal O according to δ. One pattern that we noticed among the failure cases is that they include subprograms where one shape is subtracted from another, producing a complex shape that is distant from its inputs and also distant from the final image. The second way that a benchmark can fail is that a program close to the solution is contained in the XFTA, but the Extract and Repair procedures are unable to find it. While extracting all programs accepted by the XFTA could mitigate the problem, the overhead of doing so is often prohibitively expensive.

3.6.2 Regular-Expression Synthesis

Our second application domain is regular-expression synthesis. Given a sketch and a set of positive and negative examples, the task is to find a regular expression that conforms to the given sketch and matches all positive examples, while matching none

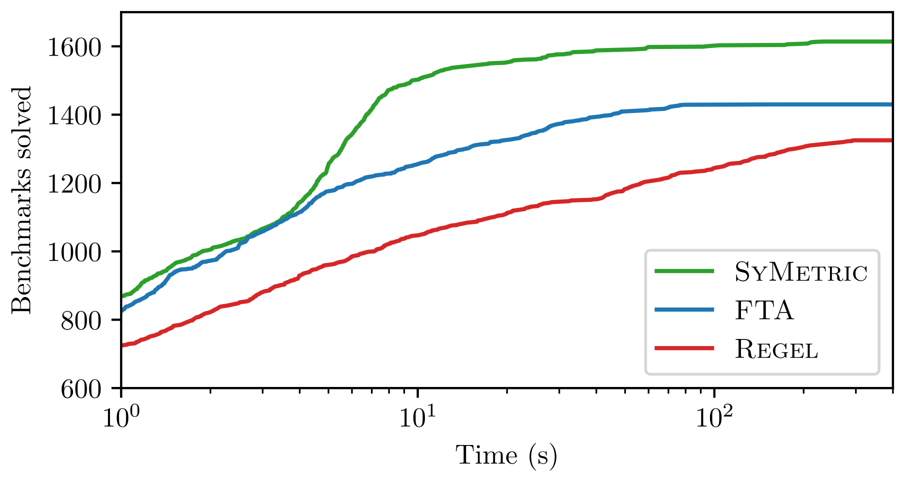


Figure 3-10: Synthesis performance on the more difficult regular-expression benchmarks.

of the negative ones.

For this application domain, we perform an empirical comparison against the following baselines:

- Regel: This baseline is a state-of-the-art regular-expression synthesis tool [15]. It performs *top-down* (rather than bottom-up) synthesis and uses several SMT-based pruning strategies to reduce the search space.
- FTA: This baseline performs bottom-up enumeration with equivalence reduction using FTAs for our regular-expression DSL.
- AFTA: As in the Inverse CSG domain, this baseline is an abstraction-based version of bottom-up enumeration with equivalence reduction [109]. We use a predicate abstraction that tracks the length of the longest match and the beginning of the first match. These predicates effectively allow the tool to avoid examining programs which do not match the whole string or that do not match from the beginning.

We note that the FTA and AFTA baselines use the same interpreter as SyMetric. However, Regel uses a different regular-expression-matching algorithm based on the Brics automaton library, which is not as efficient as our optimized implementation for this DSL.

Benchmarks For this experiment we use the Stack Overflow dataset taken from [15]. This benchmark was collected from user questions. It contains 122 distinct tasks, each of which consists of a natural-language description and a set of input-output examples. For each task, Chen et al. automatically generates a set of *sketches* (i.e. partial programs) that capture additional constraints about the target regular expression that are present in the natural-language description. This gives us a total of 2173 total task/sketch pairs to use in our evaluation. However, we note that some of these synthesis problems may not be solvable since the generated sketches could be wrong.

Experimental setup These experiments are run on a machine with two Intel Xeon 8375C processors (with a total of 128 threads) and 256 GB of RAM. We use a time limit of 5 minutes and a memory limit of 4 GB. We use $\epsilon = 0.3$, $w = 200$, and $n = 100$ for the regular-expression domain.

AFTA Instantiation

Prior work on AFTA synthesis does not consider a regular-expression domain, so we create a new predicate abstraction for regular-expression synthesis. The space of predicates contains:

- The primitive predicates $true$ and $false$.
- Equalities between constants and program results.
- $\mathsf{Longest}(k)$, which asserts that the longest match is at most k characters.
- $\mathsf{First}(k)$, which asserts the that first match starts at least k characters into the string.

Figure 3-11 gives the syntax and semantics of predicates as well as selected abstract semantics of the DSL operators.

Results summary The results of this experiment are summarized in fig. 3-9 and fig. 3-10, where the latter figure "zooms in" on the harder benchmarks. Among

$$\text{Integers } k \in \mathbb{N} \quad \text{Constants } c \in \mathsf{Match} \cup \mathbb{N}$$

$$\text{Predicates } \phi ::= \mathsf{Longest}(k) \mid \mathsf{First}(k) \mid \mathsf{Eq}(c) \mid \mathsf{True} \mid \mathsf{False}$$

$$\begin{aligned}
\textsc{Eval}_\phi(\mathsf{True}, \cdot) &= \mathit{true} \\
\textsc{Eval}_\phi(\mathsf{False}, \cdot) &= \mathit{false} \\
\textsc{Eval}_\phi(\mathsf{Eq}(c), c') &= (c = c') \\
\textsc{Eval}_\phi(\mathsf{First}(k), c) &= k \leq \min_{(i,j)\in c} i \\
\textsc{Eval}_\phi(\mathsf{Longest}(k), c) &= \max_{(i,j)\in c} j - i \leq k
\end{aligned}$$

$$\begin{aligned}
\textsc{Eval}^\sharp(f(\mathsf{Eq}(c_1), \ldots, \mathsf{Eq}(c_n))) &= \mathsf{Eq}(\textsc{Eval}(f(c_1, \ldots, c_n))) \\
\textsc{Eval}^\sharp(\mathtt{Optional}(\mathsf{True})) &= \mathsf{First}(0) \\
\textsc{Eval}^\sharp(\mathtt{And}(\mathsf{Longest}(k), \mathsf{Longest}(k'))) &= \mathsf{Longest}(\min(k, k')) \\
\textsc{Eval}^\sharp(\mathtt{And}(\mathsf{First}(k), \mathsf{First}(k'))) &= \mathsf{First}(\max(k, k')) \\
\textsc{Eval}^\sharp(\mathtt{RepeatRange}(\mathsf{Longest}(k), \mathsf{True}, \mathsf{Eq}(n))) &= \mathsf{Longest}(k \times n)
\end{aligned}$$

Figure 3-11: Syntax and semantics of predicates and selected abstract semantics of the regular-expression DSL.

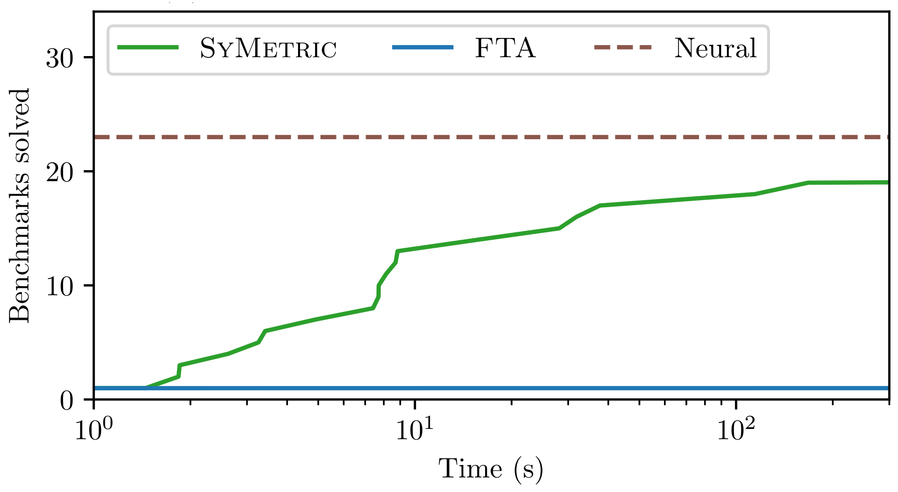


Figure 3-12: Synthesis performance on the tower-building domain.

the four tools, SyMetric achieves the best performance, solving 74% of the regex benchmarks, compared to 61% of Regel. The FTA baseline significantly outperforms the abstraction-based approach, solving 66% (for FTA) compared to 13% (for AFTA). One caveat about these results is that, while the comparison between SyMetric, FTA, and AFTA is apples-to-apples, Regel uses a different (and less efficient) interpreter for their DSL.[1] Another caveat is that, while our predicate abstraction does not yield good results, it *may* be possible to build abstractions that perform better in the regex domain. Nevertheless, we believe these results substantiate our claim that (a) SyMetric is competitive with state-of-the-art tools for the regex domain, and (b) metric program synthesis yields improvements over basic observational-equivalence reduction.

3.6.3 Solving Tower-Building Tasks

As our third application domain, we consider tower-building tasks that are inspired by planning in AI and that were used for evaluating program-synthesis tools in prior work [75, 33]. As explained in section 3.4, the goal of this task is to generate a program that constructs a given configuration of blocks.

[1]We believe it is due to this implementation difference in the interpreter that FTA slightly outperforms Regel. The pruning heuristics used in Regel allow it to scale without needing a custom interpreter.

For this domain, we compare SyMetric against two baselines:

- Neural: Our first baseline is a state-of-the-art neural synthesizer [75] that combines top-down program synthesis with a neural network that predicts the possible outputs of a partial program. Since this baseline is trained on a set of representative tower-building tasks, it can use tower motifs encountered during training to solve new tasks.
- FTA: As in the previous two domains, our second baseline performs bottom-up enumerative search with equivalence reduction using FTAs.

For this domain, we do not compare SyMetric against the abstraction-based FTA approach, as the combination of loops and mutable state make this domain difficult grounds for applying prior work [109]. In fact, we believe that applying abstraction-refinement techniques to this domain/DSL is an open research problem in its own right.

Benchmarks Our tower-building benchmarks are drawn from [75], which are constructed by systematically composing tower-building programs together (e.g., taking a program that builds a $w \times h$ bridge and building two side-by-side, varying the size and spacing). The original benchmark suite contains 40 tasks, but we found that two of the tasks are duplicates and four are not expressible in the DSL described in [75], as they require loops with variable iteration counts. We remove these six tasks, resulting in a total of 34 tower-building benchmarks used in our evaluation.

Experimental setup These experiments are performed on a machine with two Intel Xeon 8375C processors (128 threads total) and 256 GB of RAM. We use a time limit of 10 minutes and a memory limit of 4 GB. For hyperparameters, we use $\epsilon = 0.4$, $w = 100$, and $n = 100$.

Results summary The results for this domain are summarized in fig. 3-12. Note that we do not show running time for Neural, as it is not reported in [75] and we do not have access to their model. Overall, we find that SyMetric approaches the performance of the neural approach and that it performs significantly better than FTA. In particular, FTA performs poorly in this domain because the target programs tend to be fairly large and few of the enumerated programs result in the same block configuration, so equivalence reduction is not as effective in this context. SyMetric deals with the large search space size by exploiting observational similarity and by using ranking to select promising subprograms. Finally, we note that, while Neural can solve a few more benchmarks compared to SyMetric, it can do so by using motifs learned from training data as building blocks. In contrast, SyMetric can achieve similar performance without requiring access to training data.

3.6.4 Detailed Evaluation of Metric Synthesis

To gain more insights about the effectiveness of metric program synthesis and answer RQ2 and RQ3, we perform a detailed evaluation of SyMetric in the invers-CSG domain. We explore distance-based clustering in more depth and present the results of relevant ablation studies.

Ablation Studies

In this section, we describe a set of ablation studies to evaluate the relative importance of the algorithms used in our approach. We consider the following ablations:

- NoCluster: This variant does not perform clustering during FTA construction. However, it still performs repair after extracting a program from the FTA.
- NoRank: This variant does not use distance-based ranking during XFTA construction. Instead, it picks w randomly chosen (clustered) states to add to the automaton in each iteration.
- ExtractRandom: This variant does not use our proposed distance-based program-extraction technique. Instead, it randomly picks programs that are accepted by the automaton. (However, the order in which final states are considered is still determined using the distance metric.)

Benchmark	ConstructXFTA	Extract	Repair	Expansion	Clustering	Ranking
Generated	21.3/28.8	5.7/75.5	6.3/171.9	13.5/17.2	1.6/3.6	0.0/0.1
Hand-written	9.2/13.0	0.1/72.8	0.2/110.3	5.6/6.2	1.6/5.0	0.0/0.2
All	17.4/28.8	1.1/75.5	1.8/171.9	11.6/17.2	1.6/5.0	0.0/0.2

Figure 3-13: Runtime breakdown (median/max, in seconds) for sub-procedures of SyMetric and ConstructXFTA on inverse CSG.

- RepairRandom: This variant does not use our distance-based program repair technique. Instead, after applying a rewrite rule during the Repair procedure, it randomly picks one of the programs rather than using the distance metric to pick the one closest to the goal.

- SimpleDistance: This variant uses the Jaccard distance instead of the distance proposed in section 3.4.1.

The results of these ablation studies are presented in fig. 3-14 (again, for the inverse-CSG domain). For the ablations, we use the randomly generated CSG benchmarks, because they have more uniform difficulty. We find that, for this domain, the most important component of our algorithm is the distance-guided Repair procedure, followed by ranking during XFTA construction, and then clustering. The distance-guided Extract procedure has less impact, but fewer programs are synthesized if we randomly choose a program instead of using the distance metric for extraction. Finally, the distance function has a significant impact on the speed of synthesis and the number of programs solved. This supports our discussion in section 3.4.1 of the importance of local reachability.

Disabling ranking yields noticeable performance improvements for some of the easier benchmarks. While ranking is cheap on its own, if it is disabled, we only need to enumerate new states until we can build w clusters. When ranking is enabled, we need to look at the entire frontier at least once to sort it. However, ranking has a huge positive impact for the harder benchmarks and without it, the number of benchmarks solved drops significantly.

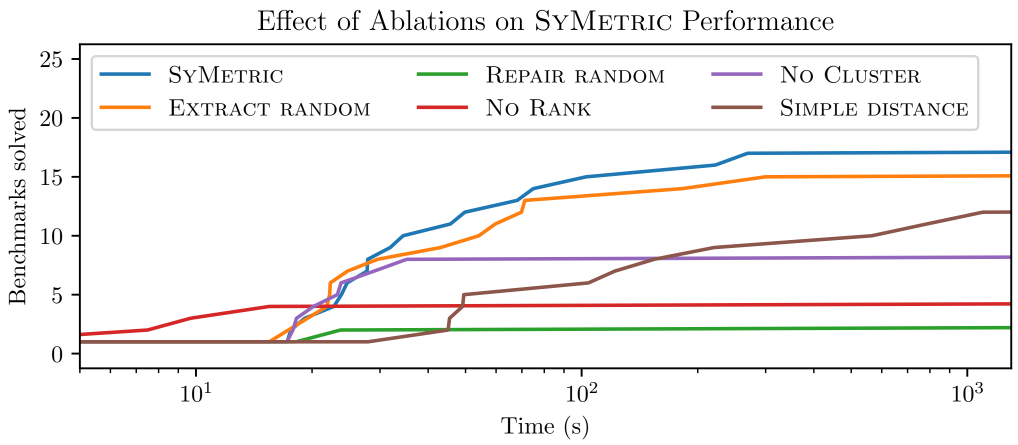


Figure 3-14: Effect of ablations on SyMetric for inverse CSG.

Detailed Evaluation of Running Time

In this section, we explore the impact of different sub-procedures on running time, again on the inverse-CSG domain. Figure 3-13 compares the running times of XFTA construction (ConstructXFTA), program extraction (Extract), program repair (Repair), and the expansion, clustering and ranking subprocedures of ConstructXFTA.

ConstructXFTA usually dominates total synthesis time. In contrast, program extraction from the XFTA using our greedy approach is quite fast, taking a median of 1.1 seconds. Finally, while the average running time of Repair is around 1.8 seconds, it varies widely depending on how many calls to Repair are made and how many rewrite rules we need to apply to find the correct program. Regarding the subprocedures of ConstructXFTA, the expansion phase dominates XFTA construction time. This is not surprising because expansion requires evaluating DSL programs to construct new states. Ranking barely takes any time, and clustering takes a median of 1.6 seconds.

3.6.5 Evaluation of the Effectiveness of Clustering

To evaluate the benefits of similarity-based clustering, we perform the following experiment on the inverse CSG domain:

1. First, we generate a set of programs with n non-terminals.

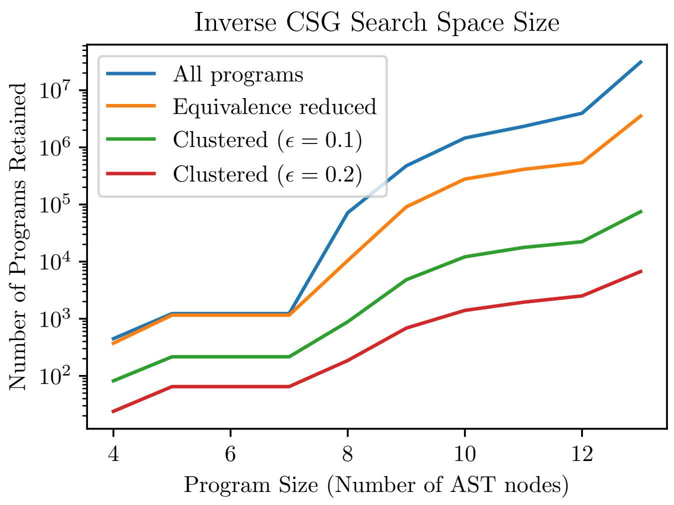


Figure 3-15: Size of the CSG program space for programs with up to n AST nodes.

2. Then, for each value of n, we apply equivalence reduction to remove equivalent programs.

3. Finally, we group the set of distinct programs using algorithm 3.

Figure 3-15 shows the number of clusters for two different values of ϵ, namely $\epsilon = 0.1$ and $\epsilon = 0.2$. We only consider programs with up to $n = 13$ AST nodes, because enumerating all programs for larger values of n is not computationally feasible.

As is evident from fig. 3-15, equivalence reduction reduces the number of programs that must be retained by approximately one order of magnitude, and similarity-based clustering reduces the search space even more dramatically. In particular, for $\epsilon = 0.1$, there is an approximately $10\times$ reduction compared to just grouping based on equivalence and an even larger reduction for the coarser ϵ value of 0.2. Hence, this experiment shows that there are many programs that are observationally similar but not equivalent, which partly explains why metric program synthesis is effective in this domain.

3.6.6 Large Language Model Case Study

Given the recent explosion of interest in applying LLMs to program synthesis, we also perform a case study applying LLMs to our evaluation domains. For each benchmark, we construct a prompt that includes a natural-language description of the DSL syntax and semantics and two examples of solved synthesis problems, consisting of program output followed by the correct program. The prompt is provided to the model and the model is queried for ten completions. We treat the problem as solved if any of the completions is correct. We run a systematic evaluation using the GPT-3.5 [77] API and we manually run several prompts through the GPT-4 [78] web interface. We do not observe a difference in performance between GPT-3.5 and GPT-4 from these manual inspections. We find that the model frequently generates syntactically well-formed programs (which is impressive considering it is presented with the DSL in the prompt) but it rarely generates correct programs.

We obtain the following results with GPT-3.5. We report the best result of the ten completions.

- **Inverse CSG:** On the 40 tasks, the model generates 0 correct and 40 well-formed programs. We experimented with two forms of prompt: a numeric prompt that represents the program output as a list of coordinates of points that are contained in the shape and an ascii-art prompt that uses a grid of characters. There was no observable difference between the two prompt forms.

- **Regular Expressions:** On the 2173 task/sketch pairs, the model generates 111 correct programs, 34 programs that match the examples but not the sketch, 1135 programs that match the sketch but not the examples, and 255 well-formed programs. The remainder are not well-formed. Many of the sketches simply assert that the program contains some subterm; the model can generate a matching program by simply repeating part of the sketch.

- **Tower Building:** On the 34 tasks, the model generates 0 correct and 34 well-formed programs.

While these results are poor, LLM performance is sensitive to the prompting strategy [93], so there may be a better prompting method that we did not discover. Multi-modal LLMs [2] may perform better on our image-based tasks. Nonetheless, we believe these results further demonstrate that the tasks in our evaluation are quite hard.

3.7 Related Work

Bottom-up Synthesis Our work builds heavily on bottom-up synthesis with equivalence reduction, an idea that Albarghouthi et al. and Udupa et al. introduced concurrently. Later work by Wang et al. explored another variation of this idea in the context of version-space learning and showed how to use Finite Tree Automata (FTA) to compactly represent the space of programs consistent with a given specification. Our work was particularly inspired by Blaze [109] which uses abstraction refinement to speed up bottom-up search. Abstractions provide a mechanism for grouping closely related solutions, allowing large sets of solutions to be ruled out by evaluating only one abstract solution. In this regard, Blaze can be viewed as performing equivalence reduction over abstract domains. In this work, we explore another relaxation of observational equivalence based on the notion of observational similarity rather than abstract equivalence. We believe that our metric-program-synthesis idea is complementary to the abstraction-refinement approach and can work well in settings for which abstract domains are hard to design or where abstractions do not effectively reduce the search space.

Quantitative Synthesis Our algorithm uses distance metrics to group similar programs and to rank them. In this regard, our method bears similarities to prior work exploring quantitative goals in program synthesis, both in the reactive-synthesis space [13] and in functional synthesis (e.g. [89, 90]). In many of these cases, the quantitative objectives are used to deal with noisy or probabilistic specifications [86, 49], where we use them to perform search more effectively.

Neural-guided Synthesis There has been significant interest in neural-guided program synthesis, where a neural network is trained to guide the search for a program that satisfies a specification. In early incarnations of this idea, the neural network was used simply to select components that were likely to be used by the program [4], but starting with the work of Devlin et al., the neural network has been heavily involved in directing the search. Especially relevant to our work is the work on execution-guided synthesis [17, 32], which uses the state of the partially constructed program to determine the most promising next step for the synthesizer. The work by Ellis et al. in particular inspired the ranking phase of our current algorithm. That work uses a learned value function—trained to evaluate the output of candidate programs—to determine which to keep as part of the beam. A common limitation of all the neural-guided-synthesis approaches is that they require significant work ahead of time to collect a dataset and train the algorithm on that dataset. In contrast, our approach relies on a domain-specific distance function, and we find that simple distance functions often work fairly well. Potential future work could apply the insights of this work in a deep-learning context.

Genetic Programming The genetic-programming community has explored the use of distance functions between observed program behaviors [70], clustering [24], and diversity [11]. Lexicase selection [52, 60] is an interesting alternative to the ranking step of our algorithm. It improves population diversity by selecting programs that perform particularly well on small subsets of the test cases.

Despite these shared ideas, our algorithm has several key differences from existing work in genetic programming. First, the focus on enumeration is a key part of our algorithm. It induces a strong bias towards short programs, which has always been a focus in the synthesis field and improves the generalization properties of synthesized programs. Second, our algorithm constructs a large, compact program space, represented as an XFTA; genetic approaches work with sets of individual programs, which increases the cost of retaining a large space. The large space is valuable during the program extraction phase. Finally, our approach is a generalization of an existing, widely used synthesis algorithm—bottom-up synthesis with equivalence reduction.

Diversity One effect of the grouping performed by our algorithm during search is to increase the diversity of the programs in the beam, allowing it to cover more distinct programs and increasing the chance a program close to the goal will be included. There has been some prior work on using diversity measures as part of search. The genetic programming community has long recognized that diversity in a population of programs is crucial to avoid converging to low-quality local minima and has explored many diversity measures to maintain diversity of the population [11]. In the context of beam search for NLP, there has been recent recognition that the top-K elements of a beam may be too close to each other and fail to capture multiple modes in the underlying distribution, which has led to the proposal of *Diverse Beam Search* to force proposals in a beam to be sufficiently different from each other [105]. Our work shares some intuitions with some of these prior works, but to our knowledge, this paper is the first to apply the idea of grouping based on a similarity function in the context of FTA-based synthesis.

Program Synthesis for Inverse CSG There has been a lot of interest in the CAD community in using program-synthesis techniques to reverse-engineer CAD problems. Two early works in this space are the work of Nandi et al. and InverseCSG [31]. Both aim to reverse-engineer 3D CSG programs from meshes, but both rely on specialized algorithms to do much of the work. Nandi et al. rely on a set of domain-specific oracles that examine the mesh and generate proposed decompositions (i.e. splitting the mesh into a union of two simpler shapes). These oracles are powerful but highly specialized to the CSG domain. Similarly, InverseCSG uses a preprocessing phase to identify all constituent primitive shapes and their parameters, so the synthesizer only has to discover the Boolean structure of the shape. In contrast, our synthesis method can solve for all primitives and their parameters without relying on a domain-specific preprocessor. InverseCSG also relies on a segmentation algorithm to break large shapes into small fragments that are then assembled into the final shape. In contrast, we aim to solve the entire inverse-CSG problem as a single synthesis task. Finally, our program space is richer than that of InverseCSG because it includes a looping construct as well as the primitives and Boolean operations. Our experimental results show that we can do with a single algorithm what in prior work required an entire pipeline of complex and specialized algorithms.

Another line of work uses neural networks to recover structured CAD models from unstructured input [92, 103]. Their performance is generally excellent but dependent on training data (e.g., Shape2Prog [103] is specialized to just furniture shapes). One of the goals of our algorithm is to only require limited domain knowledge, expressed in the distance function and repair rules. However, we believe that our approach can complement the neural methods, either by using a learned distance metric or by using a network to predict a likely space of programs, as in [63].

Further prior work processes CSG programs, either to extract common structure [56] or to capture regularity [72]. ShapeMOD [56], for example, extracts common macros from a library of CSG programs. These macros form a domain-specific language for a class of CSG programs (e.g., furniture) and help provide physically plausible results that are biased towards patterns common in the target domain. Our method does not rely on a set of training programs but could use macros like these if they were available. Nandi et al. show that it is possible to postprocess the output of an InverseCSG-like system to extract loops, which produces programs that are more general and easier to modify. However, this approach requires first synthesizing the loop-free program, which can be large for models with a lot of repetition.

Synthesis of Regular Expressions Synthesis of regular expressions from examples has a long history. Recent work includes REGEL [15] and ALPHAREGEX [62], which are top-down synthesizers that use upper and lower bounds for pruning. REGEL also uses natural-language descriptions to improve generalizability. It parses the natural-language descriptions into so-called *hierarchical sketches* and uses top-down enumerative search combined with SMT-based pruning to find a sketch completion that satisfies the examples. In our evaluation, we use the sketches generated by REGEL but solve them using metric program synthesis instead of top-down SMT-guided search. Repair of regular expressions, which is related to our local search, has also attracted significant attention from the research community [79, 87]. Because our method starts the local-search process with programs that are already fairly close to the goal, we can use simpler rewrite-based techniques for repair.

Synthesis of Tower-Building Programs The tower-building domain first appears in [33] and is used as a benchmark in [75]. The appeal of this domain comes from its loops and mutable state, as well as its connection to classical AI planning tasks. Prior work has focused on the application of neural-guided synthesis to this domain. We show that a non-neural approach can also perform well.

3.8 Conclusion

We presented a new synthesis method, called *metric program synthesis*, that performs search-space reduction using a distance metric. The key idea behind our technique is to cluster similar states during bottom-up enumeration and then perform program repair once a program that is "close enough" to the goal is found. Our approach constructs a so-called *approximate finite tree automaton* that represents a set of programs that "approximately" satisfy the specification. Our method then repeatedly extracts programs from this set and uses distance-guided rewriting to find a repair that exactly satisfies the given input-output examples.

We formalize our intuitions about which DSLs work for our proposed algorithm and show that under these (strong) conditions, our algorithms is complete. We use these conditions as a guide to instantiate our synthesis framework in three different domains (inverse CSG, regular-expression synthesis, and tower-building) by defining suitable distance metrics. Our evaluation shows that our tool, SyMetric, outperforms prior domain-agnostic FTA-based techniques in these domains. Furthermore, we compare our approach against domain-specific synthesizers and show that the performance of SyMetric is competitive with these tools despite not using any training data or domain-specific synthesis algorithms.

Chapter 4

Future Work

This section discusses future work for both Castor and SyMetric and suggests places where the two could be used together.

4.1 Extensions to Castor

There are a number of improvements that would make Castor more useful in practice.

Extensibility. There are a wide variety of physical layouts that address specific needs that Castor does not currently support. For example, bitvectors could be added to store lists of Booleans efficiently or run-length encoding could be used to store lists of scalars. Layouts which store tiles of data together and allow indexing by 2D regions could be used to store spatial data. In this work, we extend the relational algebra with a new operator for each data structure that we introduce. This approach is sufficient when the number of new structures is small, but is likely to become unwieldy as the number of structures grows. An interesting area of future work is to find ways to extend the layout algebra that increase its expressiveness while still allowing straightforward transformation rules to be written.

Integration Into Existing DBMSs. Castor does not make any attempt to integrate into an existing DBMS. This integration is an interesting opportunity for future work. Modern DBMSs support materialized views, but they are less flexible than the layouts that Castor supports, so Castor

Efficient Updates: Castor currently focuses on generating compact data structures, but these structures are not always efficiently updatable. While there are applications that use only slowly changing data, Castor would be more widely applicable if it had support for updating its generated data structures without running the compiler again. Some benefits of providing updates could be obtained by a hybrid strategy that treats the database as an append-only list or tree of blocks (similar to a log-structured merge tree [76]). New data would be appended to the end of the list as updates occur, and queries would need to read each block to determine the current state of the database. Periodic repacking would keep the length of this list under control.

Data Sharing Between Queries: Currently, a separate data structure is generated for each query in an application. These structures are small, so the space overhead is not large in practice, but it would be ideal to be able to share data structures between queries that access the same data. This sharing complicates the optimization process, because of the need to determine the granularity of sharing before optimizing the individual structures.

Reliable Optimization With E-graphs: E-graphs are a promising solution to the phase-ordering problem that appears whenever rewriting passes must be interleaved. Castor uses both heuristics and search to select a sequence of transformations, but the process is ad-hoc, and it is unclear how well it generalizes to new problems. E-graphs offer an efficient way to keep track of the results of applying rewriting rules and may make it feasible to remove some or all of the heuristics from the Castor optimizer.

4.2 Extensions to SyMetric

SyMetric is implemented using a general-purpose bottom-up enumerative synthesizer framework written in OCaml. This framework could be enhanced in several ways.

Higher-order Functions: First, the framework only supports first-order languages. Higher-order functions are difficult to work with in the context of bottom-up synthesis, because it is difficult to apply observational equivalence to eliminate equivalent lambda functions. This difficulty comes from the fact that when the function is generated, it is not known in what contexts it might be called. One possibility is to build a representation of the contexts where functions are called as the search proceeds and characterize the functions that have been found accordingly. This would allow distance metrics to be applied to programs with higher-order operators.

Learned Distance Metrics: The distance functions used in SyMetric are a potential target for machine learning. We investigated contrastive learning when building SyMetric but found that it was difficult to create high-quality distance functions. However, there are many methods for learning distance metrics, and it is likely that one of them could be applied in SyMetric.

4.3 Hybrid Deductive/Inductive Synthesis

This thesis presents both a deductive and an inductive synthesizer, but the two techniques are applied to separate applications. One area for future work is to bring inductive-synthesis techniques to bear on the data-structure-selection problem that Castor considers.

One promising area of application is in restructuring predicates. Several of Castor's transformation rules look for predicates that have particular shapes (e.g. they are conjunctive with each conjunct referring to only a single relation). Inductive synthesis could be used to sidestep this problem by directly generating an equivalent predicate of the appropriate form. Syntactic constraints are easy to apply to most inductive synthesizers, and constraint-based synthesizers can accept very expressive constraints on the solution.

Formally, the problem to solve is the following. Given a predicate ϕ, we want to find a new predicate ψ such that $\phi \equiv \psi$ and $C(\psi)$, where C is some additional syntactic or semantic constraint. Expressed as an inductive synthesis problem, we want to solve:

$$\exists \psi.\ C(\psi) \land \forall v \in \mathit{Dom}(\phi).\ \phi(v) = \psi(v).$$

While deduction rules can be used to massage a predicate into the right form, this is a tricky problem in general, and the necessary rules will depend on C. The problem is made more complex by the need to:

- Take advantage of functional dependencies or integrity constraints when reasoning about predicate equivalence. This changes the problem to:

  $$\exists \psi.\ C(\psi) \land \forall v \in \mathit{Dom}(\phi).\ F(v) \implies \phi(v) = \psi(v),$$

  where F captures the integrity constraints on the relations.

- Allow splitting a predicate into a conjunction of weaker predicates, where only one part of the conjunction satisfies C:

  $$\exists \psi, \psi'.\ C(\psi) \land \forall v \in \mathit{Dom}(\phi).\ \phi(v) = \psi(v) \land \psi'(v).$$

  This is useful when applying rules to filters, because a filter can be split into a composition of weaker filters.

Hybrid deductive/inductive synthesis has been applied to a number of other systems problems.

- StreamBit [95] is a system for generating bit-stream-manipulation code. This kind of code appears in video and image decoding, cryptography, and compression algorithms. StreamBit allows the user to provide an inefficient reference implementation of a bit-streaming pipeline. It then uses a combination of deductive and inductive synthesis to generate efficient code for running the pipeline.

- Bellmania [54] is a tool for generating dynamic-programming implementations of algorithms such as string edit distance. It uses deductive tactics to decompose the specification into subspecifications. Internally, the tactics may create and discharge proof obligations using an SMT solver. This hides the proof details from the user while preserving the semantics of the program under transformation. A synthesis tactic automatically synthesizes a program matching the specification. This tactic is applied when the problem has been sufficiently simplified by applying the deductive tactics.

Chapter 5

Conclusion

This thesis presents a deductive-synthesis approach to solving a tricky problem in database systems and a new, general-purpose inductive-synthesis tool. Program synthesis is improving at a rapid pace, driven by research from the programming-languages and machine-learning communities. Programming using a synthesis tool has become the daily reality for many software engineers. However, there are still challenges, particularly in synthesizing correct, efficient systems programs. This thesis takes a step towards a future where synthesis gives programmers the ability to quickly write programs that are correct and efficient.

Bibliography

[1] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. Weaving relations for cache performance. In Peter M. G. Apers, Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, Kotagiri Ramamohanarao, and Richard T. Snodgrass, editors, *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*, pages 169–180. Morgan Kaufmann, 2001. URL `http://www.vldb.org/conf/2001/P169.pdf`.

[2] Jean-Baptiste Alayrac, Jeff Donahue, Pauline Luc, Antoine Miech, Iain Barr, Yana Hasson, Karel Lenc, Arthur Mensch, Katherine Millican, Malcolm Reynolds, Roman Ring, Eliza Rutherford, Serkan Cabi, Tengda Han, Zhitao Gong, Sina Samangooei, Marianne Monteiro, Jacob L Menick, Sebastian Borgeaud, Andy Brock, Aida Nematzadeh, Sahand Sharifzadeh, Mikoł aj Bińkowski, Ricardo Barreira, Oriol Vinyals, Andrew Zisserman, and Karén Simonyan. Flamingo: a visual language model for few-shot learning. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pages 23716–23736. Curran Associates, Inc., 2022. URL `https://proceedings.neurips.cc/paper_files/paper/2022/file/960a172bc7fbf0177ccccbb411a7d800-Paper-Conference.pdf`.

[3] Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. Recursive program synthesis. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, pages 934–950, 2013. doi: 10.1007/978-3-642-39799-8_67. URL `https://www.microsoft.com/en-us/research/wp-content/uploads/2016/12/cav13.pdf`.

[4] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*, 2017. URL `https://openreview.net/forum?id=ByldLrqlx`.

[5] Lee Blaine, Limei Gilham, Junbo Liu, Douglas R. Smith, and Stephen J. Westfold. Planware - domain-specific synthesis of high-performance schedulers. In *The Thirteenth IEEE Conference on Automated Software Engineering, ASE 1998, Honolulu, Hawaii, USA, October 13-16, 1998*, page 270. IEEE Computer Society, 1998. doi: 10.1109/ASE.1998.732672. URL `https://doi.org/10.1109/ASE.1998.732672`.

[6] Lee Blaine, Limei Gilham, Junbo Liu, Douglas R. Smith, and Stephen J. Westfold. Planware - domain-specific synthesis of high-performance schedulers. In *The Thirteenth IEEE Conference on Automated Software Engineering, ASE 1998, Honolulu, Hawaii, USA, October 13-16, 1998*, page 270. IEEE Computer Society, 1998. ISBN 0-8186-8750-9. doi: 10.1109/ASE.1998.732672. URL `https://doi.org/10.1109/ASE.1998.732672`.

[7] Peter A. Boncz and Martin L. Kersten. MIL primitives for querying a fragmented world. *VLDB J.*, 8(2):101–119, 1999. doi: 10.1007/s007780050076. URL https://doi.org/10.1007/s007780050076.

[8] James Bornholt, Emina Torlak, Dan Grossman, and Luis Ceze. Optimizing synthesis with metasketches. In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 775–788. ACM, 2016. ISBN 978-1-4503-3549-2. doi: 10.1145/2837614.2837666. URL https://doi.org/10.1145/2837614.2837666.

[9] Fabiano C. Botelho, Rasmus Pagh, and Nivio Ziviani. Simple and space-efficient minimal perfect hash functions. In Frank K. H. A. Dehne, Jörg-Rüdiger Sack, and Norbert Zeh, editors, *Algorithms and Data Structures, 10th International Workshop, WADS 2007, Halifax, Canada, August 15-17, 2007, Proceedings*, volume 4619 of *Lecture Notes in Computer Science*, pages 139–150. Springer, 2007. doi: 10.1007/978-3-540-73951-7_13. URL https://doi.org/10.1007/978-3-540-73951-7_13.

[10] Nicolas Bruno and Surajit Chaudhuri. Automatic physical database tuning: A relaxation-based approach. In Fatma Özcan, editor, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*, pages 227–238. ACM, 2005. doi: 10.1145/1066157.1066184. URL https://doi.org/10.1145/1066157.1066184.

[11] E.K. Burke, S. Gustafson, and G. Kendall. Diversity in genetic programming: an analysis of measures and correlation with fitness. *IEEE Transactions on Evolutionary Computation*, 8(1):47–62, 2004. doi: 10.1109/TEVC.2003.819263.

[12] Rod M. Burstall and John Darlington. A transformation system for developing recursive programs. *J. ACM*, 24(1):44–67, 1977. doi: 10.1145/321992.321996. URL https://doi.org/10.1145/321992.321996.

[13] Pavol Cerný and Thomas A. Henzinger. From boolean to quantitative synthesis. In Samarjit Chakraborty, Ahmed Jerraya, Sanjoy K. Baruah, and Sebastian Fischmeister, editors, *Proceedings of the 11th International Conference on Embedded Software, EMSOFT 2011, part of the Seventh Embedded Systems Week, ESWeek 2011, Taipei, Taiwan, October 9-14, 2011*, pages 149–154. ACM, 2011. doi: 10.1145/2038642.2038666. URL https://doi.org/10.1145/2038642.2038666.

[14] Surajit Chaudhuri. An overview of query optimization in relational systems. In Alberto O. Mendelzon and Jan Paredaens, editors, *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 1-3, 1998, Seattle, Washington, USA*, pages 34–43. ACM Press, 1998. doi: 10.1145/275487.275492. URL https://doi.org/10.1145/275487.275492.

[15] Qiaochu Chen, Xinyu Wang, Xi Ye, Greg Durrett, and Isil Dillig. Multi-modal synthesis of regular expressions. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 487–502, 2020. doi: 10.1145/3385412.3385988. URL https://doi.org/10.1145/3385412.3385988.

[16] Qiaochu Chen, Aaron Lamoreaux, Xinyu Wang, Greg Durrett, Osbert Bastani, and Isil Dillig. Web question answering with neurosymbolic program synthesis. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, pages 328–343, 2021. doi: 10.1145/3453483.3454047. URL `https://doi.org/10.1145/3453483.3454047`.

[17] Xinyun Chen, Chang Liu, and Dawn Song. Execution-guided neural program synthesis. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. URL `https://openreview.net/forum?id=H1gfOiAqYm`.

[18] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. Optimizing database-backed applications with query synthesis. In Hans-Juergen Boehm and Cormac Flanagan, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 3–14. ACM, 2013. doi: 10.1145/2491956.2462180. URL `https://doi.org/10.1145/2491956.2462180`.

[19] Rada Chirkova and Michael R. Genesereth. Linearly bounded reformulations of conjunctive databases. In John W. Lloyd, Verónica Dahl, Ulrich Furbach, Manfred Kerber, Kung-Kiu Lau, Catuscia Palamidessi, Luís Moniz Pereira, Yehoshua Sagiv, and Peter J. Stuckey, editors, *Computational Logic - CL 2000, First International Conference, London, UK, 24-28 July, 2000, Proceedings*, volume 1861 of *Lecture Notes in Computer Science*, pages 987–1001. Springer, 2000. doi: 10.1007/3-540-44957-4_66. URL `https://doi.org/10.1007/3-540-44957-4_66`.

[20] Paolo Ciaccia, Marco Patella, and Pavel Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *Vldb*, volume 97, pages 426–435, 1997.

[21] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13 (6):377–387, 1970. doi: 10.1145/362384.362685. URL `http://doi.acm.org/10.1145/362384.362685`.

[22] E. F. Codd. A database sublanguage founded on the relational calculus. In E. F. Codd and A. L. Dean, editors, *Proceedings of 1971 ACM-SIGFIDET Workshop on Data Description, Access and Control, San Diego, California, USA, November 11-12, 1971*, pages 35–68. ACM, 1971.

[23] Alessandro Coglio and Cordell Green. A constructive approach to correctness, exemplified by a generator for certified Java Card applets. In Bertrand Meyer and Jim Woodcock, editors, *Verified Software: Theories, Tools, Experiments, First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10-13, 2005, Revised Selected Papers and Discussions*, volume 4171 of *Lecture Notes in Computer Science*, pages 57–63. Springer, 2005. ISBN 978-3-540-69147-1. doi: 10.1007/978-3-540-69149-5_7. URL `https://doi.org/10.1007/978-3-540-69149-5_7`.

[24] Pierre Collet, Marco Tomassini, Marc Ebner, Steven M. Gustafson, and Anikó Ekárt, editors. *Genetic Programming, 9th European Conference, EuroGP 2006, Budapest, Hungary, April 10-12, 2006, Proceedings*, volume 3905 of *Lecture Notes in Computer Science*, 2006. Springer. ISBN 3-540-33143-3. doi: 10.1007/11729976. URL `https://doi.org/10.1007/11729976`.

[25] Transaction Processing Performance Council. TPC-H benchmark specification. 21: 592–603, 2008.

[26] Philippe Cudré-Mauroux, Eugene Wu, and Samuel Madden. The case for RodentStore: An adaptive, declarative storage system. In *CIDR 2009, Fourth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2009, Online Proceedings*. www.cidrdb.org, 2009. URL `http://www-db.cs.wisc.edu/cidr/cidr2009/Paper_97.pdf`.

[27] Davi de Castro Reis, Djamel Belazzougui, Fabiano Cupertino Botelho, and Nivio Ziviani. CMPH-C minimal perfect hashing library. *C Minimal Perfect Hashing Library*, 2012.

[28] Benjamin Delaware, Clément Pit-Claudel, Jason Gross, and Adam Chlipala. Fiat: Deductive synthesis of abstract data types in a proof assistant. In Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 689–700. ACM, 2015. doi: 10.1145/2676726.2677006. URL `https://doi.org/10.1145/2676726.2677006`.

[29] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy I/O. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, pages 990–998, 2017. URL `http://proceedings.mlr.press/v70/devlin17a.html`.

[30] E. W. Dijkstra. A constructive approach to the problem of program correctness. *BIT*, 8(3):174–186, Sep 1968. ISSN 1572-9125. doi: 10.1007/bf01933419. URL `http://dx.doi.org/10.1007/bf01933419`.

[31] Tao Du, Jeevana Priya Inala, Yewen Pu, Andrew Spielberg, Adriana Schulz, Daniela Rus, Armando Solar-Lezama, and Wojciech Matusik. InverseCsg: Automatic conversion of 3d models to CSG trees. *ACM Trans. Graph.*, 37(6):213:1–213:16, 2018. doi: 10.1145/3272127.3275006. URL `https://doi.org/10.1145/3272127.3275006`.

[32] Kevin Ellis, Maxwell I. Nye, Yewen Pu, Felix Sosa, Josh Tenenbaum, and Armando Solar-Lezama. Write, execute, assess: Program synthesis with a REPL. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 9165–9174, 2019. URL `https://proceedings.neurips.cc/paper/2019/hash/50d2d2262762648589b1943078712aa6-Abstract.html`.

[33] Kevin Ellis, Catherine Wong, Maxwell I. Nye, Mathias Sablé-Meyer, Lucas Morales, Luke B. Hewitt, Luc Cary, Armando Solar-Lezama, and Joshua B. Tenenbaum. Dreamcoder: bootstrapping inductive program synthesis with wake-sleep library learning. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, pages 835–850, 2021. doi: 10.1145/3453483.3454080. URL `https://doi.org/10.1145/3453483.3454080`.

[34] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theor. Comput. Sci.*, 103(2):235–271, 1992. doi: 10.1016/0304-3975(92)90014-7. URL `https://doi.org/10.1016/0304-3975(92)90014-7`.

[35] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. Component-based synthesis of table consolidation and transformation tasks from examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 422–436, 2017. doi: 10.1145/3062341.3062351. URL `https://doi.org/10.1145/3062341.3062351`.

[36] Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. Program synthesis using conflict-driven learning. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pages 420–435, 2018. doi: 10.1145/3192366.3192382. URL `https://doi.org/10.1145/3192366.3192382`.

[37] John K. Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing data structure transformations from input-output examples. In *ACM SIGPLAN Notices*, volume 50, pages 229–239. ACM. URL `http://dl.acm.org/citation.cfm?id=2737977`.

[38] John K. Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing data structure transformations from input-output examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 229–239, 2015. doi: 10.1145/2737924.2737977. URL `https://doi.org/10.1145/2737924.2737977`.

[39] John K. Feser, Marc Brockschmidt, Alexander L. Gaunt, and Daniel Tarlow. Neural functional programming. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Workshop Track Proceedings*, 2017. URL `https://openreview.net/forum?id=Byp_ccVte`.

[40] John K. Feser, Sam Madden, Nan Tang, and Armando Solar-Lezama. Deductive optimization of relational data storage. *Proc. ACM Program. Lang.*, 4(OOPSLA): 170:1–170:30, 2020. doi: 10.1145/3428238. URL `https://doi.org/10.1145/3428238`.

[41] John K. Feser, Isil Dillig, and Armando Solar-Lezama. Metric program synthesis. In submission, 2022. URL `https://doi.org/10.48550/arXiv.2206.06164`.

[42] Fred Glover and Manuel Laguna. Tabu Search. In Ding-Zhu Du and Panos M. Pardalos, editors, *Handbook of Combinatorial Optimization: Volume1–3*, pages 2093–2229. Springer US, 1998. ISBN 978-1-4613-0303-9. doi: 10.1007/978-1-4613-0303-9_33. URL `https://doi.org/10.1007/978-1-4613-0303-9_33`.

[43] Goetz Graefe. Volcano - an extensible and parallel query evaluation system. *IEEE Trans. Knowl. Data Eng.*, 6(1):120–135, 1994. doi: 10.1109/69.273032. URL `https://doi.org/10.1109/69.273032`.

[44] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on*

Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011, pages 317–330, 2011. doi: 10.1145/1926385.1926423. URL `https://doi.org/10.1145/1926385.1926423`.

[45] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. Program synthesis. *Found. Trends Program. Lang.*, 4(1-2):1–119, 2017. doi: 10.1561/2500000010. URL `https://doi.org/10.1561/2500000010`.

[46] Himanshu Gupta, Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. Index selection for OLAP. In W. A. Gray and Per-Åke Larson, editors, *Proceedings of the Thirteenth International Conference on Data Engineering, April 7-11, 1997, Birmingham, UK*, pages 208–219. IEEE Computer Society, 1997. doi: 10.1109/ICDE.1997.581755. URL `https://doi.org/10.1109/ICDE.1997.581755`.

[47] Angélica García Gutiérrez and Peter Baumann. Modeling fundamental geo-raster operations with array algebra. In *Workshops Proceedings of the 7th IEEE International Conference on Data Mining (ICDM 2007), October 28-31, 2007, Omaha, Nebraska, USA*, pages 607–612, 2007. doi: 10.1109/ICDMW.2007.53. URL `https://doi.org/10.1109/ICDMW.2007.53`.

[48] Alon Y. Halevy. Answering queries using views: A survey. *VLDB J.*, 10(4):270–294, 2001. doi: 10.1007/s007780100054. URL `https://doi.org/10.1007/s007780100054`.

[49] Shivam Handa and Martin C. Rinard. Inductive program synthesis over noisy data. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, pages 87–98, 2020. doi: 10.1145/3368089.3409732. URL `https://doi.org/10.1145/3368089.3409732`.

[50] Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin C. Rinard, and Mooly Sagiv. Data structure fusion. In Kazunori Ueda, editor, *Programming Languages and Systems - 8th Asian Symposium, APLAS 2010, Shanghai, China, November 28 - December 1, 2010. Proceedings*, volume 6461 of *Lecture Notes in Computer Science*, pages 204–221. Springer, 2010. doi: 10.1007/978-3-642-17164-2_15. URL `https://doi.org/10.1007/978-3-642-17164-2_15`.

[51] Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin C. Rinard, and Mooly Sagiv. Data representation synthesis. In Mary W. Hall and David A. Padua, editors, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 38–49. ACM, 2011. doi: 10.1145/1993498.1993504. URL `https://doi.org/10.1145/1993498.1993504`.

[52] Thomas Helmuth, Lee Spector, and James Matheson. Solving uncompromising problems with lexicase selection. *IEEE Transactions on Evolutionary Computation*, 19(5):630–643, 2014.

[53] Jeevana Priya Inala, Rohit Singh, and Armando Solar-Lezama. Synthesis of domain specific CNF encoders for bit-vector solvers. In *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, pages 302–320, 2016. doi: 10.1007/978-3-319-40970-2_19. URL `https://doi.org/10.1007/978-3-319-40970-2_19`.

[54] Shachar Itzhaky, Rohit Singh, Armando Solar-Lezama, Kuat Yessenov, Yongquan Lu, Charles Leiserson, and Rezaul Chowdhury. Deriving divide-and-conquer dynamic programming algorithms using solver-aided transformations. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 145–164. ACM. URL `http://dl.acm.org/citation.cfm?id=2983993`.

[55] Matthias Jarke and Jürgen Koch. Query optimization in database systems. *ACM Comput. Surv.*, 16(2):111–152, 1984. doi: 10.1145/356924.356928. URL `https://doi.org/10.1145/356924.356928`.

[56] R. Kenny Jones, David Charatan, Paul Guerrero, Niloy J. Mitra, and Daniel Ritchie. Shapemod: Macro operation discovery for 3d shape programs. *ACM Trans. Graph.*, 40(4):153:1–153:16, 2021. doi: 10.1145/3450626.3459821. URL `https://doi.org/10.1145/3450626.3459821`.

[57] Fredrik Kjolstad, Shoaib Kamil, Jonathan Ragan-Kelley, David I. W. Levin, Shinjiro Sueda, Desai Chen, Etienne Vouga, Danny M. Kaufman, Gurtej Kanwar, Wojciech Matusik, and Saman P. Amarasinghe. Simit: A language for physical simulation. *ACM Trans. Graph.*, 35(2):20:1–20:21, 2016. doi: 10.1145/2866569. URL `https://doi.org/10.1145/2866569`.

[58] Yannis Klonatos, Christoph Koch, Tiark Rompf, and Hassan Chafi. Building efficient query engines in a high-level language. *Proc. VLDB Endow.*, 7(10):853–864, 2014. doi: 10.14778/2732951.2732959. URL `http://www.vldb.org/pvldb/vol7/p853-klonatos.pdf`.

[59] Tristan Knoth, Di Wang, Nadia Polikarpova, and Jan Hoffmann. Resource-guided program synthesis. In Kathryn S. McKinley and Kathleen Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 253–268. ACM, 2019. ISBN 978-1-4503-6712-7. doi: 10.1145/3314221.3314602. URL `https://doi.org/10.1145/3314221.3314602`.

[60] William La Cava, Lee Spector, and Kourosh Danai. Epsilon-lexicase selection for regression. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, pages 741–748, 2016.

[61] Tessa A. Lau, Steven A. Wolfman, Pedro M. Domingos, and Daniel S. Weld. Programming by demonstration using version space algebra. *Mach. Learn.*, 53(1-2): 111–156, 2003. doi: 10.1023/A:1025671410623. URL `https://doi.org/10.1023/A:1025671410623`.

[62] Mina Lee, Sunbeom So, and Hakjoo Oh. Synthesizing regular expressions from examples for introductory automata assignments. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2016, Amsterdam, The Netherlands, October 31 - November 1, 2016*, pages 70–80, 2016. doi: 10.1145/2993236.2993244. URL `https://doi.org/10.1145/2993236.2993244`.

[63] Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. Accelerating search-based program synthesis using learned probabilistic models. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pages 436–449, 2018. doi: 10.1145/3192366.3192410. URL `https://doi.org/10.1145/3192366.3192410`.

[64] Xiaoxuan Liu, Shuxian Wang, Mengzhu Sun, Sharon Lee, Sicheng Pan, Joshua Wu, Cong Yan, Junwen Yang, Shan Lu, and Alvin Cheung. Leveraging application data constraints to optimize database-backed web applications. *CoRR*, abs/2205.02954, 2022. doi: 10.48550/arXiv.2205.02954. URL `https://doi.org/10.48550/arXiv.2205.02954`.

[65] Calvin Loncaric, Emina Torlak, and Michael D. Ernst. Fast synthesis of fast collections. In Chandra Krintz and Emery Berger, editors, *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 355–368. ACM, 2016. doi: 10.1145/2908080.2908122. URL `https://doi.org/10.1145/2908080.2908122`.

[66] Calvin Loncaric, Michael D. Ernst, and Emina Torlak. Generalized data structure synthesis. In Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman, editors, *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 958–968. ACM, 2018. doi: 10.1145/3180155.3180211. URL `https://doi.org/10.1145/3180155.3180211`.

[67] Zohar Manna and Richard J. Waldinger. Synthesis: Dreams - programs. *IEEE Trans. Software Eng.*, 5(4):294–328, 1979. doi: 10.1109/TSE.1979.234198. URL `https://doi.org/10.1109/TSE.1979.234198`.

[68] Charith Mendis, Ajay Jain, Paras Jain, and Saman P. Amarasinghe. Revec: program rejuvenation through revectorization. In José Nelson Amaral and Milind Kulkarni, editors, *Proceedings of the 28th International Conference on Compiler Construction, CC 2019, Washington, DC, USA, February 16-17, 2019*, pages 29–41. ACM, 2019. ISBN 978-1-4503-6277-1. doi: 10.1145/3302516.3307357. URL `https://doi.org/10.1145/3302516.3307357`.

[69] Anders Miltner, Adrian Trejo Nuñez, Ana Brendel, Swarat Chaudhuri, and Isil Dillig. Bottom-up synthesis of recursive functional programs using angelic execution. *Proc. ACM Program. Lang.*, 6(POPL):1–29, 2022. doi: 10.1145/3498682. URL `https://doi.org/10.1145/3498682`.

[70] Alberto Moraglio, Krzysztof Krawiec, and Colin G Johnson. Geometric semantic genetic programming. In *Parallel Problem Solving from Nature-PPSN XII: 12th International Conference, Taormina, Italy, September 1-5, 2012, Proceedings, Part I 12*, pages 21–31. Springer, 2012.

[71] Chandrakana Nandi, James R. Wilcox, Pavel Panchekha, Taylor Blau, Dan Grossman, and Zachary Tatlock. Functional programming for compiling and decompiling computer-aided design. *Proc. ACM Program. Lang.*, 2(ICFP):99:1–99:31, 2018. doi: 10.1145/3236794. URL `https://doi.org/10.1145/3236794`.

[72] Chandrakana Nandi, Max Willsey, Adam Anderson, James R. Wilcox, Eva Darulova, Dan Grossman, and Zachary Tatlock. Synthesizing structured CAD models with equality saturation and inverse transformations. In Alastair F. Donaldson and Emina Torlak, editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 31–44. ACM, 2020. doi: 10.1145/3385412.3386012. URL `https://doi.org/10.1145/3385412.3386012`.

[73] Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. *Proc. VLDB Endow.*, 4(9):539–550, 2011. doi: 10.14778/2002938.2002940. URL `http://www.vldb.org/pvldb/vol4/p539-neumann.pdf`.

[74] Thomas Neumann and Alfons Kemper. Unnesting arbitrary queries. In Thomas Seidl, Norbert Ritter, Harald Schöning, Kai-Uwe Sattler, Theo Härder, Steffen Friedrich, and Wolfram Wingerath, editors, *Datenbanksysteme für Business, Technologie und Web (BTW), 16. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 4.-6.3.2015 in Hamburg, Germany. Proceedings*, volume P-241 of *LNI*, pages 383–402. GI, 2015. URL `https://dl.gi.de/20.500.12116/2418`.

[75] Maxwell I. Nye, Yewen Pu, Matthew Bowers, Jacob Andreas, Joshua B. Tenenbaum, and Armando Solar-Lezama. Representing partial programs with blended abstract semantics. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*, 2021. URL `https://openreview.net/forum?id=mCtadqIxOJ`.

[76] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996. doi: 10.1007/s002360050048. URL `https://doi.org/10.1007/s002360050048`.

[77] OpenAI. Introducing chatgpt, 2022. URL `https://openai.com/blog/chatgpt`.

[78] OpenAI. Gpt-4 technical report, 2023.

[79] Rong Pan, Qinheping Hu, Gaowei Xu, and Loris D'Antoni. Automatic repair of regular expressions. *Proc. ACM Program. Lang.*, 3(OOPSLA):139:1–139:29, 2019. doi: 10.1145/3360565. URL `https://doi.org/10.1145/3360565`.

[80] Matt Pharr and William R. Mark. ispc: A SPMD compiler for high-performance CPU programming. In *2012 Innovative Parallel Computing (InPar)*, pages 1–13, 2012. doi: 10.1109/InPar.2012.6339601.

[81] Rachel Pottinger and Alon Y. Levy. A scalable algorithm for answering queries using views. In Amr El Abbadi, Michael L. Brodie, Sharma Chakravarthy, Umeshwar Dayal, Nabil Kamel, Gunter Schlageter, and Kyu-Young Whang, editors, *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, pages 484–495. Morgan Kaufmann, 2000. URL `http://www.vldb.org/conf/2000/P484.pdf`.

[82] Markus Püschel, José M. F. Moura, Jeremy R. Johnson, David A. Padua, Manuela M. Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko,

Kang Chen, Robert W. Johnson, and Nicholas Rizzolo. SPIRAL: code generation for DSP transforms. *Proc. IEEE*, 93(2):232–275, 2005. doi: 10.1109/JPROC.2004.840306. URL `https://doi.org/10.1109/JPROC.2004.840306`.

[83] Markus Püschel, José M. F. Moura, Jeremy R. Johnson, David A. Padua, Manuela M. Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nicholas Rizzolo. SPIRAL: code generation for DSP transforms. *Proc. IEEE*, 93(2):232–275, 2005. doi: 10.1109/JPROC.2004.840306. URL `https://doi.org/10.1109/JPROC.2004.840306`.

[84] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman P. Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. pages 519–530, 2013. doi: 10.1145/2491956.2462176. URL `https://doi.org/10.1145/2491956.2462176`.

[85] Jonathan Ragan-Kelley, Andrew Adams, Dillon Sharlet, Connelly Barnes, Sylvain Paris, Marc Levoy, Saman P. Amarasinghe, and Frédo Durand. Halide: decoupling algorithms from schedules for high-performance image processing. *Commun. ACM*, 61 (1):106–115, 2018. doi: 10.1145/3150211. URL `https://doi.org/10.1145/3150211`.

[86] Veselin Raychev, Pavol Bielik, Martin T. Vechev, and Andreas Krause. Learning programs from noisy data. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 761–774, 2016. doi: 10.1145/2837614.2837671. URL `https://doi.org/10.1145/2837614.2837671`.

[87] Thomas Rebele, Katerina Tzompanaki, and Fabian M. Suchanek. Adding missing words to regular expressions. In *Advances in Knowledge Discovery and Data Mining - 22nd Pacific-Asia Conference, PAKDD 2018, Melbourne, VIC, Australia, June 3-6, 2018, Proceedings, Part II*, pages 67–79, 2018. doi: 10.1007/978-3-319-93037-4_6. URL `https://doi.org/10.1007/978-3-319-93037-4_6`.

[88] Tiark Rompf and Nada Amin. Functional pearl: a SQL to C compiler in 500 lines of code. In Kathleen Fisher and John H. Reppy, editors, *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*, pages 2–9. ACM, 2015. doi: 10.1145/2784731.2784760. URL `https://doi.org/10.1145/2784731.2784760`.

[89] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2013, Houston, TX, USA, March 16-20, 2013*, pages 305–316, 2013. doi: 10.1145/2451116.2451150. URL `https://doi.org/10.1145/2451116.2451150`.

[90] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic optimization of floating-point programs with tunable precision. In Michael F. P. O'Boyle and Keshav Pingali, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pages 53–64. ACM, 2014. doi: 10.1145/2594291.2594302. URL `https://doi.org/10.1145/2594291.2594302`.

[91] Amir Shaikhha, Yannis Klonatos, Lionel Parreaux, Lewis Brown, Mohammad Dashti, and Christoph Koch. How to architect a query compiler. In Fatma Özcan, Georgia Koutrika, and Sam Madden, editors, *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 1907–1922. ACM, 2016. doi: 10.1145/2882903.2915244. URL https://doi.org/10.1145/2882903.2915244.

[92] Gopal Sharma, Rishabh Goyal, Difan Liu, Evangelos Kalogerakis, and Subhransu Maji. Csgnet: Neural shape parser for constructive solid geometry. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*, pages 5515–5523, 2018. doi: 10.1109/CVPR.2018.00578. URL http://openaccess.thecvf.com/content_cvpr_2018/html/Sharma_CSGNet_Neural_Shape_CVPR_2018_paper.html.

[93] Chenglei Si, Zhe Gan, Zhengyuan Yang, Shuohang Wang, Jianfeng Wang, Jordan Lee Boyd-Graber, and Lijuan Wang. Prompting GPT-3 to be reliable. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=98p5x51L5af.

[94] Armando Solar-Lezama. Program sketching. *Int. J. Softw. Tools Technol. Transf.*, 15(5-6):475–495, 2013. doi: 10.1007/s10009-012-0249-7. URL https://doi.org/10.1007/s10009-012-0249-7.

[95] Armando Solar-Lezama, Rodric M. Rabbah, Rastislav Bodík, and Kemal Ebcioglu. Programming by sketching for bit-streaming programs. In Vivek Sarkar and Mary W. Hall, editors, *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, pages 281–294. ACM, 2005. ISBN 1-59593-056-6. doi: 10.1145/1065010.1065045. URL https://doi.org/10.1145/1065010.1065045.

[96] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. Combinatorial sketching for finite programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006*, pages 404–415, 2006. doi: 10.1145/1168857.1168907. URL https://doi.org/10.1145/1168857.1168907.

[97] Michael Stonebraker. The choice of partial inversions and combined indices. *International Journal of Parallel Programming*, 3(2):167–188, 1974. doi: 10.1007/BF00976642. URL https://doi.org/10.1007/BF00976642.

[98] Michael Stonebraker. SciDB: An open-source DBMS for scientific data. *ERCIM News*, 2012(89), 2012. URL http://ercim-news.ercim.eu/en89/special/scidb-an-open-source-dbms-for-scientific-data.

[99] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alex Rasin, Nga Tran, and Stanley B. Zdonik. C-store: A column-oriented DBMS. In Klemens Böhm, Christian S. Jensen, Laura M. Haas, Martin L. Kersten, Per-Åke Larson, and Beng Chin Ooi, editors, *Proceedings of the*

31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005, pages 553–564. ACM, 2005. URL `http://www.vldb.org/archives/website/2005/program/paper/thu/p553-stonebraker.pdf`.

[100] Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *ACM Trans. Embed. Comput. Syst.*, 13 (4s):1–25, 2014. doi: 10.1145/2584665. URL `https://doi.org/10.1145/2584665`.

[101] Ruby Y. Tahboub, Grégory M. Essertel, and Tiark Rompf. How to architect a query compiler, revisited. In Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 307–322. ACM, 2018. doi: 10.1145/3183713.3196893. URL `https://doi.org/10.1145/3183713.3196893`.

[102] Zohreh Asgharzadeh Talebi, Rada Chirkova, Yahya Fathi, and Matthias F. Stallmann. Exact and inexact methods for selecting views and indexes for OLAP performance improvement. In Alfons Kemper, Patrick Valduriez, Noureddine Mouaddib, Jens Teubner, Mokrane Bouzeghoub, Volker Markl, Laurent Amsaleg, and Ioana Manolescu, editors, *EDBT 2008, 11th International Conference on Extending Database Technology, Nantes, France, March 25-29, 2008, Proceedings*, volume 261 of *ACM International Conference Proceeding Series*, pages 311–322. ACM, 2008. doi: 10.1145/1353343.1353383. URL `https://doi.org/10.1145/1353343.1353383`.

[103] Yonglong Tian, Andrew Luo, Xingyuan Sun, Kevin Ellis, William T. Freeman, Joshua B. Tenenbaum, and Jiajun Wu. Learning to infer and execute 3d shape programs. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*, 2019. URL `https://openreview.net/forum?id=rylNH20qFQ`.

[104] Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M.K. Martin, and Rajeev Alur. Transit: Specifying protocols with concolic snippets. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 287–296, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2014-6. doi: 10.1145/2491956.2462174. URL `http://doi.acm.org/10.1145/2491956.2462174`.

[105] Ashwin K. Vijayakumar, Michael Cogswell, Ramprasaath R. Selvaraju, Qing Sun, Stefan Lee, David J. Crandall, and Dhruv Batra. Diverse beam search for improved description of complex scenes. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, pages 7371–7379, 2018. URL `https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/17329`.

[106] Eelco Visser. A survey of strategies in rule-based program transformation systems. *J. Symb. Comput.*, 40(1):831–873, 2005. doi: 10.1016/j.jsc.2004.12.011. URL `https://doi.org/10.1016/j.jsc.2004.12.011`.

[107] Chenglong Wang, Alvin Cheung, and Rastislav Bodík. Synthesizing highly expressive SQL queries from input-output examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 452–466, 2017. doi: 10.1145/3062341.3062365. URL `https://doi.org/10.1145/3062341.3062365`.

[108] Xinyu Wang, Isil Dillig, and Rishabh Singh. Synthesis of data completion scripts using finite tree automata. *Proceedings of the ACM on Programming Languages*, 1 (OOPSLA):1–26, 2017.

[109] Xinyu Wang, Isil Dillig, and Rishabh Singh. Program synthesis using abstraction refinement. *Proc. ACM Program. Lang.*, 2(POPL):63:1–63:30, 2018. doi: 10.1145/3158151. URL `https://doi.org/10.1145/3158151`.

[110] Karl D. D. Willis, Yewen Pu, Jieliang Luo, Hang Chu, Tao Du, Joseph G. Lambourne, Armando Solar-Lezama, and Wojciech Matusik. Fusion 360 gallery: a dataset and environment for programmatic CAD construction from human design sequences. *ACM Trans. Graph.*, 40(4):54:1–54:24, 2021. doi: 10.1145/3450626.3459818. URL `https://doi.org/10.1145/3450626.3459818`.

[111] Cong Yan and Alvin Cheung. Generating application-specific data layouts for inmemory databases. *Proc. VLDB Endow.*, 12(11):1513–1525, 2019. doi: 10.14778/3342263.3342630. URL `http://www.vldb.org/pvldb/vol12/p1513-yan.pdf`.

[112] Kuat Yessenov, Ivan Kuraj, and Armando Solar-Lezama. Demomatch: API discovery from demonstrations. In Albert Cohen and Martin T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 64–78. ACM, 2017. doi: 10.1145/3062341.3062386. URL `https://doi.org/10.1145/3062341.3062386`.