Program Synthesis for Data Structure Refactoring

James Dong

May 21, 2021

Abstract

Data structure refactoring is a common task for many programmers today, but it requires significant amounts of error-prone manual effort to change all of the code that relies on it. In this paper, I present two techniques for automating data structure refactoring using program synthesis techniques. Program synthesis allows programmers to automatically synthesize programs according to a specification, and the synthesized program is guaranteed to conform to the specification, leaving no room for human error. The first technique, Migrator, completely automates the task of rewriting SQL queries after a schema migration, given only the source and destination schemas. The second technique, Solidare, automates the task of rewriting a Solidity program given a set of data structure transformations. Both of these techniques allow a programmer to transform an entire program with minimal effort and guaranteed-correct results.

Contents

1 Introduction **3**

2 Background: Partial Weighted MaxSAT **3**

3 Database Schema Migration with Migrator **5**
- 3.1 Background . . . 6
- 3.2 Overview . . . 7
  - 3.2.1 Value correspondence generation . . . 8
  - 3.2.2 Sketch generation . . . 9
  - 3.2.3 Sketch completion . . . 9
- 3.3 Definition of Equivalence for Database Programs . . . 10
- 3.4 Synthesis Algorithm . . . 12
  - 3.4.1 Lazy enumeration of value correspondences . . . 12
  - 3.4.2 Sketch generation . . . 13
- 3.5 Sketch Completion . . . 16
- 3.6 Soundness and Completeness . . . 18
- 3.7 Evaluation . . . 18

4 Smart Contract Optimization with Solidare **21**
4.1 Background . . . 21
4.1.1 Types . . . 23
4.1.2 Type expressions . . . 23
4.2 Overview . . . 24
4.2.1 Motivation and usage scenario . . . 24
4.2.2 Transformation of type declarations . . . 24
4.2.3 Transformation of variable declarations . . . 24
4.2.4 Sketch generation . . . 25
4.3 DSL Description . . . 26
4.3.1 Type aliases . . . 26
4.3.2 Syntax . . . 27
4.4 DSL Semantics . . . 27
4.4.1 Semantics over structure definitions . . . 28
4.4.2 Semantics over variable types . . . 29
4.5 Optimal Synthesis of Smart Contracts . . . 31
4.5.1 Sketch generation . . . 31
4.5.2 Sketch completion . . . 34
4.6 Implementation . . . 37
4.7 Sketch Generation . . . 37
4.8 Sketch Completion . . . 38
4.9 Equivalence Checking . . . 38
4.10 Generating Minimal Failing Subcontracts . . . 38
4.11 Soundness and Completeness . . . 38
4.12 Evaluation . . . 38

5 Conclusion **42**

6 Acknowledgements **42**

1 Introduction

How data is laid out is a critical part of all software. For a variety of reasons, including performance, maintainability, and feature addition, a program's data layout can undergo many changes throughout its lifetime; this is true for many types of applications, including databases and smart contracts. However, even a small change in a program's data structure can require changing large swaths of code that touches it, often in non-trivial ways. Furthermore, it is very easy to make a mistake during this refactoring, which can lead to unexpected and sometimes subtle bugs. Nonetheless, this error-prone task must be done whenever a programmer wants or needs to change a program's data layout.

The task of data structure migration is challenging for various reasons. First, since a piece of data can be accessed in several places throughout the code, a programmer must locate and manually fix all of them. For large codebases, this factor alone can cause a potential refactoring to be completely infeasible by hand. Next, it can be very difficult and time-consuming to replace code after a refactoring task, and it is not always possible to verify the correctness of a fix looking only at its local context. Also, when performing refactoring for performance reasons, a programmer typically must do so multiple times to find the most performant data layout. Since a single refactoring is already a slow task, this type of refactoring can be extremely time-consuming.

One solution that can significantly alleviate the burden on programmers for performing this task is program synthesis. Program synthesis tools take a given high-level specification and automatically generate a program that is correct according to the specification. Program synthesis has already been applied to a wide variety of domains, including general synthesis of functional programs[3], data-wrangling tasks[2], and text processing in spreadsheets[5].

For data structure refactoring tasks, this means that along with the original program, programmers only need to provide a specification of how the data is to be rearranged, and the tool can automatically synthesize the resulting program. In this paper, I present two techniques in which program synthesis is used to automate data structure refactoring tasks: Migrator and Solidare.

2 Background: Partial Weighted MaxSAT

In both tools, we reduce certain enumeration problems to *partial weighted MaxSAT*. To solve these MaxSAT instances, we use the Sat4J[6] library.

Partial weighted MaxSAT is a generalization of the boolean satisfiability problem (SAT), in which in addition to traditional "hard" constraints, there are also *soft constraints* (ψ, w) which are not required to be satisfied but are associated with a weight w. The goal of the problem is to maximize the sum of the weights of the soft constraints while ensuring the hard constraints are still satisfied. More formally, a MaxSAT problem is a set of hard constraints H and soft constraints S where the goal is to find an

interpretation I such that:

$$I \models \bigwedge_{\phi \in H} \phi, \text{ and}$$

$$\sum_{\substack{(\psi, w) \in S \\ I \models \psi}} w \quad \text{is maximized.}$$

Most SAT solvers, including Sat4J, only accept formulas in conjunctive normal form, or CNF. We can use arbitrary formulas by first translating them into CNF; this can be done in linear time using a procedure known as *Tseitin's transformation*, in which auxiliary variables are introduced to represent the results of sub-expressions. This can be expressed as the following:

$$\frac{v \text{ variable}}{v \rightsquigarrow_{\mathsf{Ts}} (v, \top)} \text{ (Tseitin-Var)} \quad \frac{\phi \rightsquigarrow_{\mathsf{Ts}} (v, \psi) \quad v' \text{ fresh}}{\bar{\phi} \rightsquigarrow_{\mathsf{Ts}} (v', (v \vee v') \wedge (\bar{v} \vee \bar{v}') \wedge \psi)} \text{ (Tseitin-Neg)}$$

$$\frac{\phi_1 \rightsquigarrow_{\mathsf{Ts}} (v_1, \psi_1) \quad \phi_2 \rightsquigarrow_{\mathsf{Ts}} (v_2, \psi_2) \quad v' \text{ fresh}}{\phi_1 \wedge \phi_2 \rightsquigarrow_{\mathsf{Ts}} (v', (\bar{v}_1 \vee \bar{v}_2 \vee v') \wedge (v_1 \vee \bar{v}') \wedge (v_2 \vee \bar{v}') \wedge \psi_1 \wedge \psi_2)} \text{ (Tseitin-And)}$$

$$\frac{\phi_1 \rightsquigarrow_{\mathsf{Ts}} (v_1, \psi_1) \quad \phi_2 \rightsquigarrow_{\mathsf{Ts}} (v_2, \psi_2) \quad v' \text{ fresh}}{\phi_1 \vee \phi_2 \rightsquigarrow_{\mathsf{Ts}} (v', (v_1 \vee v_2 \vee \bar{v}') \wedge (\bar{v}_1 \vee v') \wedge (\bar{v}_2 \vee v') \wedge \psi_1 \wedge \psi_2)} \text{ (Tseitin-Or)}$$

Then if $\phi \rightsquigarrow_{\mathsf{Ts}} (v, \phi')$, Tseitin's transformation $\mathsf{Ts}(\phi) = v \wedge \phi'$.

A key property of this transformation is that the original variables are preserved and the satisfying assignments of ϕ and $\mathsf{Ts}(\phi)$ are in one-to-one correspondence, which means that enumerating the solutions of one gives the same solutions as the other.

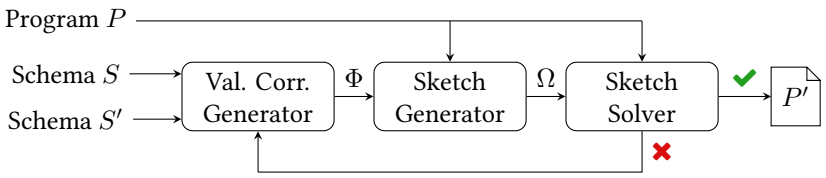

Figure 1: MIGRATOR methodology

3 Database Schema Migration with MIGRATOR

Many applications today rely on a relational database to store their data. For example, most contemporary web applications use SQL databases to store site and user data and generate webpages dynamically. As a result, databases play a critical role in many web applications, and it is important that these applications are able to interface with databases to retrieve the correct data and prevent data loss and corruption.

Databases often undergo *schema refactoring*, in which the schema, or structure, of the database is changed. This can be for various reasons, such as improving performance, implementing new features, or simplifying how the data is laid out. However, changing the database schema requires that all code that interfaces with the database be rewritten to conform to the new schema. Especially for structural refactorings that change data layout across tables, this is a non-trivial task, which makes it a prime candidate for using program synthesis.

In order to solve this problem, we have created a synthesis tool called MIGRATOR[12] which automatically performs a database migration given the source program and source and destination schemas[1]. Our synthesis methodology is illustrated in Figure 1. Notice that instead of synthesizing the output program directly, we break down this task into three simpler subtasks.

First, we generate a *value correspondence* Φ between the source and destination schemas. The value correspondence is a list of mappings between source and destination columns[8]. There are two main reasons why we generate a value correspondence as an intermediate step instead of encoding it in the sketch. First, the value correspondence is relatively easy to guess based on column names and types. Second, for a given value correspondence, the search space of possible programs is dramatically decreased, and the generated sketch is guaranteed to be consistent about how columns are mapped.

After this, we generate a *program sketch* Ω that encodes all possible candidate programs using the value correspondence Φ. A program sketch is a program where some values are unknown (known as *holes*) and must be filled in to form a concrete program. This process is done by applying a series of rules to the original program based on Φ.

Finally, we "solve" the sketch by filling in the holes, or instantiating it, in such a way that the resulting program P' is equivalent to the original program P. Sketch solving is done via enumerative search with some optimizations, particularly the use

[1]Source code available at https://github.com/utopia-group/migrator

of *minimum failing inputs*, to guide the search process and prune unproductive instantiations. One advantage of this approach over traditional approaches such as counterexample-guided inductive synthesis (CEGIS) that use SMT solvers to reason about program semantics is that encoding the semantics of relational algebra into quantifier-free first-order theories supported by SMT solvers is difficult, both to encode and solve.

We have evaluated our tool on a dataset of 20 database programs collected from textbooks and popular open-source repositories; MIGRATOR is able to solve all 20 benchmarks with an average synthesis time of 69.4 seconds per benchmark. We also show that in particular, MIGRATOR is significantly (over 5x in all benchmarks) faster than SKETCH[10], a tool which uses the CEGIS approach.

3.1 Background

Most databases today are relational databases, in which queries are expressed using relational algebra. Relational databases store data in the form of *tables*, or relations, which are made up of *columns*, or attributes, which specify the structure of the data in the table. The data in a table is made up of *rows*, or tuples of data. Furthermore, for each table there is at most one column which is designated as the *primary key* (PK). The primary key is required to be unique for each row, and a row is how data in other tables is linked to a specific row.

Data in a relational database is retrieved using *queries*, which are specified using relational algebra. Relational algebra uses the basic set operations from set theory (such as union, difference, and Cartesian product) along with a few core operations: projection, selection, rename, and joins.

Projection, denoted $\Pi_{a_1,\ldots,a_n}(R)$, where R represents the relation to be projected and $a_1, \ldots, a_n$ a set of attribute names, represents the result when the tuples of R are restricted to $\{a_1, \ldots, a_n\}$. Selection, denoted $\sigma_\phi(R)$, where ϕ is a proposition formula on the attributes of R, represents the set of tuples of R where ϕ is true. Rename, denoted $\rho_{a/b}(R)$, renames the attribute b to a in all tuples of R.

Although there are many types of joins in relational algebra, the most common type of join is an equijoin, here denoted $R \,{}_a\!\bowtie_b S$, where a is an attribute of R and b an attribute of S. We furthermore restrict our attention to the common case of equijoins on *foreign keys*, which are columns that are specified to correspond to the primary key of another table and are considered part of the database schema. The semantics of an equijoin $R \,{}_a\!\bowtie_b S$ is equivalent to $\sigma_{a=b}(R \times S)$, that is, the set of all pairs of tuples from R and S such that a and b have the same value. In the case of equijoins on foreign keys, this is equivalent to appending the data from the unique row specified by the foreign key. For convenience, when there is only one foreign key applicable b corresponding to primary key a, we use the shorthand $R \bowtie S$ for $R \,{}_a\!\bowtie_b S$. We also use the term *join chain* to refer to the relation formed by a chain of joins as well as the database tables involved.

To represent insertions into a table, we use the notation $\texttt{ins}(T, \{a_1 : v_1, \ldots, a_n : v_n\})$. For column updates, we use the notation $\texttt{upd}(T, \phi, a, v)$, which sets the attribute a to the value v for all rows where ϕ is true. Finally, the notation $\texttt{del}([T], T, \phi)$ deletes from T all rows where ϕ is true.

In order to simplify presentation, in this paper, we allow updates on *virtual tables* in the form of join chains across multiple tables. The semantics of these virtual updates is straightforward: an insertion simply inserts into all participating tables, an update updates the singular table corresponding to the attribute to be updated, and a deletion deletes from the set of tables specified by the first parameter: for example, $\texttt{del}([T_1, T_2], T_1 \bowtie T_2 \bowtie T_3, \phi)$ deletes only from tables T_1 and T_2, even though the join chain $T_1 \bowtie T_2 \bowtie T_3$ and predicate ϕ may be specified over all three tables. As a caveat, when new primary and foreign keys are introduced to the schema, unique values are generated for those columns when not specified.

Most database programs, instead of using relational algebra directly, use a language called SQL to represent queries and updates. SQL queries take a fixed form: first, the columns to be projected, or the symbol $*$, representing all columns, is given following the keyword **SELECT**, followed by the tables involved, using the **FROM** keyword and the **JOIN** ... **ON** keywords, and finally a predicate is given using the **WHERE** keyword. The SQL query

$$\begin{aligned}&\textbf{SELECT } a_1, \ldots, a_n \textbf{ FROM } T_1\\ &\qquad\qquad\textbf{JOIN } T_2 \textbf{ ON } a_2 = b_2\\ &\qquad\qquad\ldots\\ &\qquad\qquad\textbf{JOIN } T_m \textbf{ ON } a_m = b_m \textbf{ WHERE } \phi\end{aligned}$$

is semantically equivalent to the relational algebra expression

$$\Pi_{a_1,\ldots,a_n}(\sigma_\phi(T_1 \ {}_{a_2}\!\bowtie_{b_2} T_2 \cdots {}_{a_m}\!\bowtie_{b_m} T_m)).$$

Insertions, updates, and deletes are similar:

$$\begin{aligned}&\textbf{INSERT INTO } T \textbf{ VALUES } (v_1, \ldots, v_n)\\ &\textbf{UPDATE } T \textbf{ JOIN}^2 \ \ldots \textbf{ SET } a_1 = v_1, \ldots, a_n = v_n \textbf{ WHERE } \phi\\ &\textbf{DELETE } T_1, \ldots, T_n \textbf{ FROM } T \textbf{ JOIN}^2 \ \ldots \textbf{ WHERE } \phi\end{aligned}$$

correspond to, respectively:

$$\begin{gathered}\texttt{ins}(T, \{a_1 : v_1, \ldots, a_n : v_n\}), \text{ where } a_i \text{ is the } i\text{th column of } T\\ \texttt{upd}(T \bowtie \ldots, \phi, a_1, v_1); \ldots; \texttt{upd}(T, \phi, a_n, v_n), \text{ performed simultaneously}\\ \texttt{del}([T_1, \ldots, T_n], T \bowtie \ldots, \phi).\end{gathered}$$

3.2 Overview

In this section, we give an overview of our synthesis technique using a simple motivating example. Consider the database program given in Figure 2. Since it is expensive to fetch a potentially large binary image every time information about an instructor or TA is requested, the programmer wishes to consolidate instructor and TA pictures into one table, Picture. The destination schema is also given in Figure 4.

[2] **JOIN**s inside **UPDATE**/**DELETE** statements is not part of standard SQL, but it is supported in many SQL implementations. As it is can be converted automatically to a subquery with the same semantics, we use this feature for brevity in this paper. Similarly, multiple **DELETE** is nonstandard but used so that there is a one-to-one correspondence with $\texttt{del}(\ldots)$ notation.

Schema:

Class(ClassId, InstId, TaId)
Instructor(InstId, IName, IPic)
TA(TaId, TName, TPic)

update addInstructor(*int* id, *string* name, *binary* pic)
    INSERT INTO Instructor **VALUES** (id, name, pic)
update deleteInstructor(*int* id)
    DELETE FROM Instructor **WHERE** InstId = id
query getInstructorInfo(*int* id)
    SELECT IName, IPic **FROM** Instructor **WHERE** InstId = id
update addTA(*int* id, *string* name, *binary* pic)
    INSERT INTO TA **VALUES** (id, name, pic)
update deleteTA(*int* id)
    DELETE FROM TA **WHERE** TaId = id
query getTAInfo(*int* id)
    SELECT IName, IPic **FROM** TA **WHERE** TaId = id

Figure 2: Example database program

3.2.1 Value correspondence generation

The first thing MIGRATOR does is lazily enumerate possible value correspondences. As mentioned in the introduction, a value correspondence describes how columns in the source schema are mapped to the destination schema. More formally, a value correspondence[3] Φ between source schema S and destination schema S' is a mapping from each attribute in S to a *set* of attributes in S'. If for some tables $T \in S, T' \in S'$ and attributes $a \in T, a' \in S'$, we have $T'.a' \in \Phi(T.a)$, this means that $T'.a'$ holds the same data as $T.a$. If $\Phi(T.a) = \emptyset$, this means that the column was deleted in the new schema, and if $|\Phi(T.a)| > 1$, this means that the column was duplicated. Lazy enumeration is done using a MAXSAT encoding, using heuristics to prefer more "natural" correspondences.

For the example program, the first value correspondence enumerated (which happens to be the correct one) contains mappings between identical columns and two additional mappings:

$$\text{Instructor.IPic} \mapsto \text{Picture.Pic}$$
$$\text{TA.IPic} \mapsto \text{Picture.Pic}$$

[3]We use a slightly simplified version of the notion of value correspondences as presented in [8]. For example, their notion of value correspondence also allows for columns to be mapped to the value of a function applied to other columns. Our approach can be extended to allow this but at the cost of expanding the search space.

update addInstructor(*int* id, *string* name, *binary* pic)
    INSERT INTO $??_1\{$Picture $\bowtie$ Instructor, Picture $\bowtie$ TA $\bowtie$ Instructor,
        Picture $\bowtie$ TA $\bowtie$ Class $\bowtie$ Instructor$\}$ **VALUES** (id, name, pic)
update deleteInstructor(*int* id)
    DELETE $??_2\{$[Picture], . . . , [Picture, Instructor, TA, Class]$\}$ **FROM** $??_3\{$
        Picture $\bowtie$ Instructor, Picture $\bowtie$ TA $\bowtie$ Instructor,
        Picture $\bowtie$ TA $\bowtie$ Class $\bowtie$ Instructor$\}$ **WHERE** InstId $=$ id
query getInstructorInfo(*int* id)
    SELECT IName, Pic **FROM** $??_4\{$Picture $\bowtie$ Instructor,
        Picture $\bowtie$ TA $\bowtie$ Instructor,
        Picture $\bowtie$ TA $\bowtie$ Class $\bowtie$ Instructor$\}$ **WHERE** InstId $=$ id
update addTA(*int* id, *string* name, *binary* pic)
    INSERT INTO $??_5\{$Picture $\bowtie$ TA, Picture $\bowtie$ Instructor $\bowtie$ TA,
        Picture $\bowtie$ Instructor $\bowtie$ Class $\bowtie$ TA$\}$ **VALUES** (id, name, pic)
update deleteTA(*int* id)
    DELETE $??_6\{$[Picture], . . . , [Picture, Instructor, TA, Class]$\}$ **FROM** $??_7\{$
        Picture $\bowtie$ TA, Picture $\bowtie$ Instructor $\bowtie$ TA,
        Picture $\bowtie$ Instructor $\bowtie$ Class $\bowtie$ TA$\}$ **WHERE** InstId $=$ id
query getTAInfo(*int* id)
    SELECT IName, Pic **FROM** $??_8\{$Picture $\bowtie$ TA, Picture $\bowtie$ Instructor $\bowtie$ TA,
        Picture $\bowtie$ Instructor $\bowtie$ Class $\bowtie$ TA$\}$ **WHERE** InstId $=$ id

Figure 3: Program sketch for Figure 2 and first value correspondence

3.2.2 Sketch generation

Given a value correspondence Φ, the next step is to generate a program sketch Ω that encodes the set of candidate programs that are consistent with Φ. This is done by a series of rules based on the source program statements and the value correspondence.

The generated sketch is shown in Figure 3. This sketch, although the source program is very simple, already has a very large number of possible completions: 164,025 exactly.

3.2.3 Sketch completion

Finally, we complete the sketch Ω by instantiating its holes. As mentioned in the introduction, solving this sketch using traditional techniques based on SMT is difficult due to the complex semantics of SQL. We deal with this by reverting to enumerative search but using clever pruning techniques such as minimum failing inputs to reduce the search space.

We perform enumerative search by using a SAT encoding. We begin by only encoding that each hole has one completion: for example, for hole $??_1$, we generate the following constraint[4]:

$$\Psi_1 = \oplus(b_1^1, b_1^2, b_1^3)$$

[4]The SAT solver we use, Sat4J[6], supports exactly-one constraints natively.

where b_i^j represents a boolean variable encoding that hole $??_i$ has been assigned the jth completion and $\oplus$ is the n-ary xor (i.e. exactly one input is true). The final constraint is then

$$\Psi = \Psi_1 \wedge \cdots \wedge \Psi_8.$$

MIGRATOR then passes this constraint to the SAT solver, which returns a model. Suppose the model returned is as follows:

$$M = b_1^3 \wedge b_2^2 \wedge b_3^3 \wedge b_4^3 \wedge b_5^1 \wedge b_6^4 \wedge b_7^3 \wedge b_8^3.$$

Unfortunately, this model does not correspond to a program which is equivalent to the original program. As a result, we must *block* this assignment by appending a *blocking clause* to the constraint Ψ. A blocking clause ϕ for an assignment M must ensure that the same assignment is never generated again: in particular, it must hold that $\phi \implies \neg M$. A naive way to do this would be to simply append the clause $\neg M$, but this approach would require 164,025 iterations in order to fully explore the search space.

To generate smarter blocking clauses, we instead compute a *minimum failing input*, which is a shortest sequence of function calls such that result of the sequence in P is different from that of P'. For example, the following sequence ω is a minimum failing input for the assignment M:

$$\omega = \text{addTA(1, “Aditya Tewari”, } \texttt{89504E470D0A1A01}\ldots\text{); getTAInfo(1)}$$

The output of the original program is the single row (“Aditya Tewari”, `89504E47`…), while the output of the new program is empty.

Let H be the set of holes (here $\{??_5, ??_8\}$) involved in the functions called by ω. Then observe that changing any combination of holes *not* in H while keeping the holes in H the same would not change the result of ω at all. Therefore, we know that we must change at least one hole in H in order to change the output of ω and thus be possibly valid. From this insight, we obtain the stronger blocking clause $\neg(b_5^1 \wedge b_8^3)$, which eliminates 18,255 possible instantiations.

Finally, after a few more iterations, MIGRATOR obtains the following correct model:

$$M^* = b_1^1 \wedge b_2^2 \wedge b_3^1 \wedge b_4^1 \wedge b_5^1 \wedge b_6^4 \wedge b_7^1 \wedge b_8^1,$$

which corresponds to the program P' in Figure 4. After running a final verification step, MIGRATOR returns the program P' as the synthesis result.

3.3 Definition of Equivalence for Database Programs

The notion of equivalence used here is the same as prior work focused on verifying equivalence of database programs[11].

For a database program P over schema S with update functions $\{U_1, \ldots, U_n\}$ and query functions $\{Q_1, \ldots, Q_m\}$, an *invocation sequence* for P has the form

$$\omega = (f_1, \sigma_1); \ldots; (f_{k-1}, \sigma_{k-1}); (f_k, \sigma_k)$$

Schema:

Class(ClassId, InstId, TaId)
Instructor(InstId, IName, PicId)
TA(TaId, TName, PicId)
Pic(PicId, Pic)

update addInstructor(*int* id, *string* name, *binary* pic)
INSERT INTO Instructor **VALUES** (id, name, u_1)
INSERT INTO Picture **VALUES** (u_1, pic)
update deleteInstructor(*int* id)
DELETE Instructor**FROM** Picture **JOIN** Instructor
ON Picture.PicId = Instructor.PicId **WHERE** InstId = id
query getInstructorInfo(*int* id)
SELECT IName, Pic **FROM** Instructor **JOIN** Picture
ON Instructor.PicId = Picture.PicId **WHERE** InstId = id
update addTA(*int* id, *string* name, *binary* pic)
INSERT INTO TA **VALUES** (id, name, u_1)
INSERT INTO Picture **VALUES** (u_1, pic)
update deleteTA(*int* id)
DELETE TA **FROM** Picture **JOIN** TA
ON Picture.PicId = TA.PicId **WHERE** TaId = id
query getTAInfo(*int* id)
SELECT IName, Pic **FROM** TA **JOIN** Picture
ON TA.PicId = Picture.PicId **WHERE** TaId = id

Figure 4: Refactored version of Figure 2

Algorithm 1 Synthesizing database programs

1: **procedure** SYNTHESIZE(P, S, S')
Input: source program P over source schema S, target schema S'
Output: target program P' or $\perp$, indicating failure
2: **loop**
3: $\Phi \leftarrow$ NEXTVALUECORR(S, S')
4: **if** $\Phi = \perp$ **then return** $\perp$
5: $\Omega \leftarrow$ GENSKETCH(Φ, P)
6: $P' \leftarrow$ COMPLETESKETCH(Φ, P)
7: **if** $P \neq \perp$ **then return** P'

where f_1 through f_{k-1} are update functions and f_k is a query function, and σ_i contains the arguments for function f_i. Note that each invocation sequence has only one update and it is at the end; this is because updates after the last query do not have any effect on the output, and any invocation sequence with multiple queries can be split up to form multiple invocation sequences with the same effect, and so when considering equivalence we need only worry about a single query at the end. We use the notation $[\![P]\!]_\omega$ to denote the result of executing the sequence ω in program P starting with an empty database.

For two database programs P, P' with the same function signatures, we say that P is equivalent to P', denoted $P \simeq P'$, if for all invocation sequences ω, we have $[\![P]\!]_\omega = [\![P']\!]_\omega$.

3.4 Synthesis Algorithm

The overall synthesis algorithm is given in Algorithm 1. In this section, I will describe the components of the synthesis algorithm in more detail.

3.4.1 Lazy enumeration of value correspondences

In order to guarantee completeness of our synthesis algorithm, we enumerate all possible value correspondences between the source and destination schemas. As doing so eagerly is not practical, we lazily enumerate the possible value correspondences using MAXSAT, as mentioned in the introduction.

Variables. For all attributes $a_1, \ldots, a_n$ in the source schema and $a'_1, \ldots, a'_m$ in the destination schema, we introduce a boolean variable x_i^j, representing that in the value correspondence Φ, attribute a_i is mapped to attribute a_j in the destination schema. That is,

$$x_i^j \iff a_j \in \Phi(a_i).$$

Hard constraints. There are two types of hard constraints, which rule out value correspondences which are guaranteed to not correspond to any valid programs:

Type compatibility: If two attributes a_i and a_j' have different types, they cannot possibly be mapped to each other. This is encoded using the following constraint:

$$\bigwedge_{i,j} \text{type}(a_i) \neq \text{type}(a'_j) \rightarrow \neg x_i^j.$$

Existence of queried attributes: For every attribute a_i that is queried in the source program, $\Phi(a_i)$ must be nonempty, i.e. it must be mapped to some a'_j in the destination schema as otherwise there would be no way to store or retrieve the data in that attribute. This implies the following constraint:

$$\bigwedge_{i} \text{queried}(a_i) \rightarrow \bigvee_{j} x_i^j.$$

Soft constraints. In addition to the hard constraints, we also use two types of soft constraints to capture heuristics about what value correspondences are more likely to be correct. First, since attributes are more likely to be renamed to something which is similar to the original name, we would like to prioritize value correspondences where similarly-named attributes are mapped to each other. To do this, using the following definition of similarity, where lev is the Levenshtein edit distance and $\text{sim}_{\max}$ is a constant equal to 100:

$$\text{sim}(T.a, T'.a') \coloneqq \text{sim}_{\max} - \text{lev}(a, a') - \frac{1}{2}\text{lev}(T, T'),$$

we introduce the following soft constraints:

$$\bigwedge_{i,j} (x_i^j, \text{sim}(a_i, a'_j)).$$

Additionally, since one-to-one mappings are more common than one-to-many mappings, we introduce the following constraints, where w_{dup} is a constant equal to 100:

$$\bigwedge_{i} \bigwedge_{1 \leq j \leq m} \bigwedge_{j < k \leq m} (x_i^j \rightarrow \neg x_i^k, w_{\text{dup}}).$$

3.4.2 Sketch generation

Given a value correspondence Φ as enumerated in the previous section, we now generate a *program sketch* encoding all possible programs that may be equivalent to the source program P under Φ.

Sketch language. In addition to the standard constructs as described in the background section, a program sketch may contain two *nondeterministic* constructs that must be filled in to obtain a concrete program. First, a program sketch can contain *holes*, denoted $??\{e_1, \ldots, e_n\}$, where the set $\{e_1, \ldots, e_n\}$ is referred to as the *domain* of that hole. A hole can be completed by substituting it with a member of its domain.

$$\frac{A \subseteq \text{Attrs}(J) \quad \forall a \in A.\ \exists a' \in \Phi(a).\ a' \in \text{Attrs}(J')}{\Phi \vdash_A J \sim J'} \text{ (Attrs)}$$

$$\frac{A = \text{Attrs}(J) \quad \Phi \vdash_A J \sim J'}{\Phi \vdash J \sim J'} \text{ (JoinChain)}$$

Figure 5: Inference rules for join correspondences

Additionally, a program sketch can contain the *choice* construct[5] $s_1 \parallel s_2$, which is shorthand for the conditional statement

$$\textbf{if ??}\{\top, \bot\} \textbf{ then } s_1 \textbf{ else } s_2.$$

Join correspondence. In order to reason about how join chains are mapped between the source and destination schemas, we generate a set of *join correspondences* between join chains in the source program and possible join chains in the destination schema. The inference rules for possible join correspondences is listed in Figure 5. Specifically, in order for a join correspondence (J, J') to be valid given a set of attributes A, denoted $\Phi \vdash_A J \sim J'$, J', Φ must map every attribute in A to some attribute in J'. Similarly, the notation $\Phi \vdash J \sim J'$ means that Φ maps *every* attribute in J to some attribute in J'. Furthermore, observe that join correspondences are not unique: if $\Phi \vdash J \sim J'_1$ and $\Phi \vdash J \sim J'_2$, this means that join chain J could be mapped to *either* J'_1 or J'_2 in the target program.

Sketch generation. The rules for sketch generation are summarized in Figure 6. Most of these rules are straightforward: the base cases Attr and Join simply rewrite join chains and attributes using the given value correspondence, and the Filter, Proj, Update, and Insert rules rewrite the nested attributes, predicates, and queries recursively. The Delete rule requires a bit more explanation: to delete a row in a join chain, it is sufficient to delete the corresponding rows from *any* non-empty subset of the tables involved. Therefore, the domain for the set of tables to delete from is equal to the powerset of the tables in the join minus the empty set. However, notice that these rules are not deterministic, as there are multiple possible join chains in the join correspondence induced by Φ.

Sketch composition. To resolve this ambiguity, we *compose* all possible sketches to obtain a more general sketch. We use the notation $\Phi \vdash S \twoheadrightarrow \Omega$ to denote this kind of judgment. We begin with a single sketch, using the following rule:

$$\frac{\Phi \vdash S \rightsquigarrow \Omega}{\Phi \vdash S \twoheadrightarrow \Omega} \text{ (Lift)}$$

[5] In the original paper, we use the notation $s_1 \textcircled{?} s_2$ instead.

$$\frac{\Phi \vdash J \sim J'}{\Phi \vdash J \rightsquigarrow J'}\ \text{(Join)} \qquad \frac{\Phi(a) = \{a'_1, \ldots, a'_n\}}{\Phi \vdash a \rightsquigarrow \texttt{??}\{a'_1, \ldots, a'_n\}}\ \text{(Attr)}$$

$$\frac{a_1, \ldots, a_n = \mathrm{Attrs}(\phi) \quad \forall i.\ \Phi \vdash a_i \rightsquigarrow h_i}{\Phi \vdash \phi \rightsquigarrow \phi[h_1/a_1, \ldots, h_n/a_n]}\ \text{(Pred)}$$

$$\frac{\Phi \vdash Q \rightsquigarrow \Omega \quad \Phi \vdash \phi \rightsquigarrow \phi'}{\Phi \vdash \sigma_\phi(Q) \rightsquigarrow \sigma_{\phi'}(\Omega)}\ \text{(Filter)}$$

$$\frac{\begin{array}{c}\Phi \vdash Q(J) \rightsquigarrow \Omega(J') \quad \forall i.\ \Phi \vdash a_i \rightsquigarrow h_i \\ A = \{a_1, \ldots, a_n\} \cup \mathrm{Attrs}(Q) \quad \Phi \vdash_A J \sim J'\end{array}}{\Phi \vdash \Pi_{a_1,\ldots,a_n}(Q(J)) \rightsquigarrow \Pi_{h_1,\ldots,h_m}(\Omega(J'))}\ \text{(Proj)}$$

$$\frac{\begin{array}{c}A = \mathrm{Attrs}(L) \cup \mathrm{Attrs}(\phi) \quad \Phi \vdash \phi \rightsquigarrow \phi' \\ \Phi \vdash_A J \sim J' \quad 2^{\mathrm{Tables}(J')} - \emptyset = \{L_1, \ldots, L_n\}\end{array}}{\Phi \vdash \texttt{del}(L, J, \phi) \rightsquigarrow \texttt{del}(\texttt{??}\{L_1, \ldots, L_n\}, J', \phi}\ \text{(Delete)}$$

$$\frac{\begin{array}{c}\Phi \vdash \phi \rightsquigarrow \phi' \quad \Phi \vdash a \rightsquigarrow h \\ A = \mathrm{Attrs}(\phi) \cup \mathrm{Attrs}(\{a\}) \quad \Phi \vdash_A J \sim J'\end{array}}{\Phi \vdash \texttt{upd}(J, \phi, a, v) \rightsquigarrow \texttt{upd}(J', \phi', h, v)}\ \text{(Update)}$$

$$\frac{\Phi \vdash J \sim J' \quad \forall i.\ \Phi \vdash a_i \rightsquigarrow h_i}{\Phi \vdash \texttt{ins}(J, \{a_i : v_i, \ldots\}) \rightsquigarrow \texttt{ins}(J', \{h_j : v_j, \ldots\})}\ \text{(Insert)}$$

Figure 6: Inference rules for sketch generation

Next, we combine queries by using the choice operator:

$$\frac{\Phi \vdash Q \twoheadrightarrow \Omega \quad \Phi \vdash Q \rightsquigarrow \Omega' \quad \Omega = \Omega_1 \parallel \ldots \parallel \Omega_n \quad \forall i.\ \Omega' \neq \Omega_i}{\Phi \vdash Q \twoheadrightarrow \Omega \parallel \Omega'} \text{ (Query)}$$

For updates (`ins`, `upd`, `del`), we must account for the possibility that for any two updates Ω'_1, Ω'_2, either one or *both* of the updates may happen. Thus we use the following rule:

$$\frac{\Phi \vdash U \twoheadrightarrow \Omega \quad \Phi \vdash U \rightsquigarrow \Omega' \quad \Omega = \Omega_1 \parallel \ldots \parallel \Omega_n \quad \forall i.\ \Omega' \neq \Omega_i}{\Phi \vdash U \twoheadrightarrow \Omega \parallel \Omega' \parallel \Omega \cdot \Omega'} \text{ (Update)}$$

where the notation $U_1 \cdot U_2$ is defined as follows:

$$\begin{aligned} U_1 \cdot U_2 &= U_1; U_2 \\ (U_1; U_2) \cdot U_3 &= U_1; U_2; U_3 \\ (U_1 \parallel U_2) \cdot U_3 &= (U_1 \cdot U_3) \parallel (U_2 \cdot U_3) \end{aligned}$$

Then the final sketch is obtained by obtaining the rules in this section to a fixed point.

Enumeration of join correspondences. During sketch generation, we must enumerate all possible join correspondences in order to produce the most general sketch. We do this algorithmically by analogy to the Steiner tree problem on a graph where the vertices are tables and the edges represent possible joins.

In more detail, for source and destination schemas S, S', consider the graph $G = (V, E)$, where $V = \text{Tables}(S')$ and $E = \{(T, T') \mid T \text{ and } T' \text{ can be joined}\}$, where two tables can be joined if one contains a foreign key corresponding to the other table's primary key, as described in the background section. Then a join chain can be viewed as a connected subgraph of G, where the edges represent joins. In particular, ignoring cyclic joins and self-joins[6], we need only concern ourselves with the case of join chains whose graph representation is a tree. In order for the resulting join correspondence for a subgraph $C = (V', E')$ to be valid, we must also require that for every attribute a_i in the attribute set A, there exists some attribute $a'_i \in \Phi(a_i)$ in a vertex T such that $T \in V'$.

In our implementation, we do this by generating all possible spanning trees and then pruning leaves until no more leaves can be removed without invalidating the join correspondence.

3.5 Sketch Completion

After generating a sketch Ω, we now perform enumerative search over the set of programs encoded by Ω. The basic algorithm for sketch completion is outlined in Algorithm 2.

[6]They do not appear in the benchmarks, so we did not include this in our implementation, but they can be handled by duplicating the graph and adding edges according to the cyclic structure of the join chain in the source program.

Algorithm 2 Sketch completion

```
1:  procedure CompleteSketch(Ω, P)
    Input: sketch Ω, source program P
    Output: target program P′ or ⊥, indicating failure
2:      Ψ ← Encode(Ω)
3:      while SAT(Ψ) do
4:          M ← GetModel(Ψ)
5:          P′ ← Instantiate(Ω, M)
6:          if Verify(P, P′) then
7:              return P′
8:          ω ← MinCex(P, P′)
9:          Ψ ← Ψ ∧ Block(M, ω)
10:     return ⊥
```

Initial SAT encoding. The first step in sketch completion is generating a boolean formula Ψ that encodes all possible instantiations of the sketch Ω. This is done by introducing a constraint for each hole that it must be filled by exactly one element of its domain:

$$\Psi = \bigwedge_{??_i \in \text{Holes}(\Omega)} \oplus(b_i^1, \ldots, b_i^{n_i}),$$

where b_i^j is a boolean variable representing that the ith hole has been filled with the jth element of its domain.

Verification and generating minimum failing inputs. Since invoking Mediator[11] to perform verification is very expensive, we instead perform exhaustive testing up to a bound on the number of statements. Additionally, this gives us the ability to retrieve minimal counterexamples (precise definition given in the next section) for each failed verification result[7]. In detail, to generate all invocation sequences with n statements, for statements 1 to $n-1$, we generate all possible updates with parameters given by a seed set of constants C, and for statement n we generate all possible queries using the same set of parameters. Then for each invocation sequence ω we check if the result given by programs P and P' are the same. If they are different, we return ω as the minimum failing input. Otherwise, we pass the programs onto Mediator to decide equivalence.

Blocking clause generation. As discussed in the overview, we aim to block as many possible invocations as possible given a minimum failing input. A minimum failing input for programs P and P' is defined as an invocation sequence ω such that

$$[\![P]\!]_\omega \neq [\![P']\!]_\omega, \text{ and}$$
$$\nexists\, \omega'.\ |\omega'| < |\omega| \wedge [\![P]\!]_{\omega'} \neq [\![P']\!]_{\omega'}.$$

[7] Note that if exhaustive testing fails to find a counterexample but Mediator decides that the programs are not equivalent, we do not obtain a minimum failing input. However, this does not seem to occur in practice.

Observe that if H is the set of holes in functions that appear in ω (and H^{c} is the set of holes that are not in H), any program that instantiates the same values to the holes in H, regardless of the assignments to H^{c}, is *guaranteed* to produce the same result for ω. Furthermore, since ω is a *minimum* failing input, the size of H^{c} is maximized, so we are able to block more assignments than other failing inputs.

To encode this as a blocking clause, we add the following constraint, where k_i is the index of the assignment to hole i:

$$\phi = \neg(\bigwedge_{??_i \in H} b_i^{k_i})$$

3.6 Soundness and Completeness

As shown in the appendix included in [13], which is the extended version of [12], MIGRATOR is both sound, which means that generated programs are guaranteed to be equivalent to the original program, and relatively complete, which means that if there exists some program $P' \simeq P$ which is *structurally isomorphic* to the original program P, MIGRATOR is guaranteed to find that program, with some soundness and completeness assumptions on the verifier (note that MEDIATOR, the verifier we use, is both sound and relatively complete[11]). Structural isomorphism, which is also defined in the appendix, roughly means that two programs share the same general structure and are related to each other according to a value correspondence.

3.7 Evaluation

We have evaluated MIGRATOR on a dataset of 20 programs taken from textbooks, online tutorials, and real-world web applications available on GitHub, taken from prior work[11]. The results are summarized in Table 1. See [12] for a detailed description of the experimental setup and explanation of the results table as well as descriptions of the refactoring operation for each benchmark.

We can see from the results that MIGRATOR is able to synthesize all 20 benchmarks in an average of 69.4 seconds per benchmark, or 1.2 seconds per function. This indicates that our tool can be very useful in automatically performing schema refactoring, especially given that it requires no additional user input.

Comparison with SKETCH. Table 2 compares our tool with SKETCH[10], which uses the SMT-driven CEGIS approach to generate candidate programs. As shown in the table, SKETCH is unable to synthesize all real-world benchmarks and two textbook benchmarks as well as being significantly slower (5.3x to 10455.0x) than MIGRATOR. This demonstrates that our proposed approach is able to scale better than the standard CEGIS approach for this domain.

Table 1: Main experimental results.

	Benchmark	Funcs	Source		Target		VCs	Iters	Synth Time(s)	Total Time(s)
			Tbl	Att	Tbl	Att				
textbook bench	Oracle-1	4	2	8	1	6	1	1	0.3	2.7
	Oracle-2	19	3	17	7	25	1	5	0.5	11.3
	Ambler-1	10	1	6	2	7	1	2	0.3	2.9
	Ambler-2	10	2	7	1	6	1	1	0.3	0.6
	Ambler-3	7	2	5	2	5	2	5	0.4	30.6
	Ambler-4	5	1	2	1	2	1	1	0.3	0.5
	Ambler-5	8	2	5	3	6	5	7	0.3	3.1
	Ambler-6	10	2	9	2	8	1	1	0.3	0.7
	Ambler-7	8	2	7	2	8	1	1	0.3	0.6
	Ambler-8	14	3	10	3	13	1	7	0.5	3.1
real-world bench	cdx	138	16	125	17	131	1	7	11.9	38.9
	coachup	45	4	51	5	55	1	10	1.8	6.7
	2030Club	125	15	155	16	159	1	2	5.2	24.8
	rails-ecomm	65	8	69	9	75	1	6	2.5	10.3
	royk	151	19	152	19	155	1	17	46.1	60.1
	MathHotSpot	54	7	38	8	42	6	11	1.2	5.8
	gallery	58	7	52	8	57	1	11	2.5	9.4
	DeeJBase	70	10	92	11	97	1	8	3.5	9.3
	visible-closet	263	26	248	27	252	1	108	1304.7	1370.8
	probable-engine	85	12	83	11	78	1	9	4.6	17.5
	Average	**57.5**	**7.2**	**57.1**	**7.8**	**59.4**	**1.5**	**11.0**	**69.4**	**80.5**

Table 2: Comparison with Sketch.

	Benchmark	Sketch	
		Synth Time(s)	**Speedup**
textbook bench	Oracle-1	88.2	294.0x
	Oracle-2	>86400.0	>172800.0x
	Ambler-1	3136.5	10455.0x
	Ambler-2	71.5	238.3x
	Ambler-3	74.7	186.8.5x
	Ambler-4	1.6	5.3x
	Ambler-5	494.4	1648.0x
	Ambler-6	226.2	754.0x
	Ambler-7	814.8	2716.0x
	Ambler-8	>86400.0	>172800.0x
real-world bench	cdx	>86400.0	>7260.5x
	coachup	>86400.0	>48000.0x
	2030Club	>86400.0	>16615.4x
	rails-ecomm	>86400.0	>34560.0x
	royk	>86400.0	>1874.2x
	MathHotSpot	>86400.0	>72000.0x
	gallery	>86400.0	>34560.0x
	DeeJBase	>86400.0	>24685.7x
	visible-closet	>86400.0	>66.2x
	probable-engine	>86400.0	>18782.6x
	Average	**>52085.4**	**>750.5x**

4 Smart Contract Optimization with Solidare

The Ethereum blockchain has seen rapid growth, with currently over $367 billion worth of Ethereum in the blockchain[9]. One feature of Ethereum is that it lets users create *smart contracts*, which are programs that run on the blockchain and can perform trades automatically. Since smart contracts require the user to pay money (*gas*) in order to deploy, it is critical that they are optimized in terms of gas efficiency.

Although prior work (e.g. [1]) has focused on optimization techniques for reducing gas usage of smart contracts, they do not consider optimizations that change the underlying *data layout* of programs, which could lead to substantial gas savings. For example, consider the example contract and its refactored version as shown in Figure 7. Here the refactored version uses a different layout of structures to store the same data, which leads to gas savings of approximately 30%. However, determining exactly what data structure leads to the most gas-efficient contract is very difficult without extensive knowledge of the Solidity compiler and EVM, as these kinds of changes can produce counter-intuitive results.

Therefore, our tool Solidare[4] allows the user to specify the desired refactoring using a simple DSL that acts on the data types of the program. The tool then automatically transforms the program's data structures and variables according to the specification and synthesizes an equivalent program automatically. As a result, programmers can easily experiment with different transformations and compare their gas usage.

The architecture of Solidare is similar to that of Migrator, but without the value correspondence generation step, as this is already known based on the provided transformation description. First, we generate a *contract sketch* that encodes all possible well-typed contract implementations that are consistent with the given transformation. As in Migrator, this is done by applying a series of rules to the program.

Next, we solve the sketch by using an *optimal program synthesis* technique that finds a completion to the sketch that is both equivalent to the original program and minimal in terms of gas usage (with respect to a proxy gas model). The sketch completion procedure is similar to Migrator as well, using *minimal failing subcontracts* to prune the search space.

We have evaluated our tool on a dataset of 20 real-world smart contracts found in Etherscan; Solidare is able to reduce their gas usage up to 48.6% with an average synthesis time of 31.4 seconds per benchmark. We also show that in Solidare with the optimal synthesis technique improves upon a baseline using enumerative search by about 31% in the number of contracts solved.

4.1 Background

For the purposes of this paper, a smart contract is a program that runs on the Ethereum[14] blockchain. Most smart contracts are written in Solidity, which contains features that specifically target the Ethereum Virtual Machine (EVM). Smart contracts are executed by *miners* who receive a fee for performing computations, so someone who wants to deploy a smart contract on the blockchain must pay a fee (measured in *gas*) that represents the computational cost of the program. As a result, smart contract

```
contract CreditDAO {
  struct Election {
    address maxVotes;
    uint nextCandidateIndex;
    mapping(address => bool) candidates;
    mapping(address => bool) userHasVoted;
    mapping(uint => uint) canVotes;
    uint maxVotes;
    uint idProcessed;
  }
  ...
  uint public nextIdx;
  mapping(uint => Election) public elections;

  function submitForElection() {
    elections[nextIdx-1].nextCandidateIndex++;
    elections[nextIdx-1].candidates[msg.sender] = true;
  }
  function vote() {
    elections[nextIdx-1].canVotes[candidateId] += 1;
    elections[nextIdx-1].userHasVoted[msg.sender] = true;
  }
  function finishElections(uint _iterations) {
    uint currentVotes;
    uint nextId = elections[nextIdx-1].idProcessed;
    for (uint cnt = 0; cnt < _iterations; cnt++) {
      currentVotes = elections[nextIdx-1].canVotes[nextId];
      if (currentVotes > elections[nextIdx-1].maxVotes) {
        elections[nextIdx-1].maxVotes = currentVotes;
      }
    }
  }
}
```

```
contract CreditDAOT {
  struct Election {
    address maxVotes;
    uint nextCandidateIndex;
    mapping(address => Candidate) candidates;

    mapping(uint => uint) canVotes;
  }
  struct Count {
    uint maxVotes;
    uint idProcessed;
  }
  struct Candidate {
    bool candidates;
    bool voted;
  }
  ...
  uint public nextIdx;
  mapping(uint => Election) public elections;
  mapping(uint => Count) public counts;

  function submitForElection() {
    elections[nextIdx-1].nextCandidateIndex++;
    elections[nextIdx-1].candidates[msg.sender].candidates = true;
  }
  function vote() {
    elections[nextIdx-1].canVotes[candidateId] += 1;
    elections[nextIdx-1].candidates[msg.sender].voted = true;
  }
  function finishElections(uint _iterations) {
    uint currentVotes;
    uint nextId = counts[nextIdx-1].idProcessed;
    for (uint cnt = 0; cnt < _iterations; cnt++) {
      currentVotes = elections[nextIdx-1].canVotes[nextId];
      if (currentVotes > counts[nextIdx-1].maxVotes) {
        counts[nextIdx-1].maxVotes = currentVotes;
      }
    }
  }
}
```

Figure 7: Motivating example program

programmers invest significant effort into optimizing their programs for minimal gas usage. From looking at the gas usage of several programs, the most significant factor that contributes to gas usage in most programs is accessing *blockchain variables*, which are much more expensive to access than local variables. As a result, in our tool SOLIDARE we focus on transforming a program's data structures in order to minimize accesses to blockchain variables as well as minimizing the number of statements executed in general.

In this paper, we represent a program P as a tuple $(\Sigma, \Gamma, V, \mathbf{F})$ where

- Σ is a *structure environment*, mapping the names of structures in the program to their definition, which is a tuple of types $(\tau_1, \ldots, \tau_n)$.
- Γ is a *type environment*, mapping variables (both fields and local variables) to their types. The notation $\Gamma \vdash e : \tau$ means that under the typing environment Γ, expression e has type τ.
- $V \subseteq \mathrm{dom}(\Gamma)$ is the set of blockchain variables.
- $\mathbf{F}$ is a set of functions that can be invoked by Ethereum users. Function bodies consist of statements, including assignments, loads, stores, conditionals, and loops.

4.1.1 Types

There are three basic kinds of types in Solidity, namely primitive types (such as *uint* and *address*), structures, and mappings.

Structures. A structure S is a named tuple $(\tau_1, \ldots, \tau_n)$, where τ_i denotes the type of the ith field in S. As in Solidity, We use the notation $S(e_1, \ldots, e_n)$ to construct a new value of type S where the value of the ith field is e_i. Structures can either be stored as a value using the **memory** keyword or as a reference. In this paper, we will assume all structures are stored by reference as value structures can be treated as a special case.

Mappings and arrays. The type $\textbf{mapping}(W \Rightarrow \tau)$ represents a key-value store where the key has type W, which must be a primitive type, and the value has type τ. Mappings are always stored as a reference. Note that arrays are a special case of mappings where the key type is *uint*.

4.1.2 Type expressions

In this paper, we use the notation ξ to denote a *type expression* with a single hole. We use the notation $\xi(\tau)$ to represent filling in that hole with the type τ. For example, if ξ is $\textbf{mapping}(\textit{uint} \Rightarrow ?)$, then $\xi(\textit{address})$ is $\textbf{mapping}(\textit{uint} \Rightarrow \textit{address})$.

In particular, when used in inference rules, the notation $\xi(\tau)$ means that the rule applies to any type that contains τ as a subterm.

We also use the notation $\xi(\tau_1, \ldots, \tau_n)$ to mean $(\xi(\tau_1), \ldots, \xi(\tau_n))$.

4.2 Overview

4.2.1 Motivation and usage scenario

Suppose a smart contract developer wants to optimize the smart contract shown on the left in Figure 7. As mentioned in the introduction, the refactoring shown significantly reduces gas usage for this contract. This refactoring can be expressed in our DSL using two transformations: first, some fields in the Election structure are moved to a new structure, Count. Next, a new structure called Candidate is introduced and encapsulates the two boolean fields in the Election structure. In our DSL, this is written as follows:

$$\text{Election, Count} = \textbf{Split}(\text{Election}, 5);$$
$$\text{Candidate} = \textbf{Wrap}(\textit{bool}, \textit{bool});$$

This reduces gas usage for the following reasons:

1. Since the fields maxVotes and idProcessed are not accessed as frequently as the other fields, placing them in a separate structure helps avoid unnecessary reads from the blockchain.
2. By introducing a new Candidate structure with two booleans, we can merge the two mappings used in the Election structure into a single mapping. This transformation ends up reducing the number of read and write operations on the blockchain.

It is important to note that *both* of these transformations are necessary to reduce the contract's gas usage, as either of the two do not yield any significant gas savings in isolation. In practice, we found that the impact of a given transformation is very difficult and counterintuitive to predict beforehand. Therefore, with Solidare we aim to provide programmers with tools to quickly try out different transformations and observe their impact on gas usage.

4.2.2 Transformation of type declarations

Given a source program $P = (\Sigma, \Gamma, V, \mathbf{F})$ (refer to the background section for an explanation on what these symbols mean) and a transformation $\mathbf{T}$, the first thing Solidare does is generate new data types based no $\mathbf{T}$. In our example, it generates three structures: Election, Count, and Candidate, as shown in lines 2–15 on the right side of Figure 7. Observe also that the fields candidates and userHasVoted have been merged into a single field candidates.

4.2.3 Transformation of variable declarations

Next, Solidare modifies and introduces variable declarations as necessary. For our example, it does not change any existing declarations but adds the following new variable (renamed for clarity):

$$\textbf{mapping}(\textit{uint} \Rightarrow \text{Counts})\ \text{counts};$$

```
function submitForElection() {
  elections[nextIdx − 1].nextCandidateIndex++;
  if (??1{true, false})
    ??2{elections[nextIdx − 1].candidates[msg.sender].candidate, . . .} = true;
}
function vote(uint candidateId) {
  elections[nextIdx − 1].candidateVotes[candidateId] += 1;
  if (??3{true, false})
    ??4{elections[nextIdx − 1].candidates[msg.sender].voted, . . .} = true;
}
function finishElections(uint _iterations) {
  uint currentVotes; uint nextId;
  if (??5{true, false})
    nextId = ??6{true, false};
  for (uint cnt = 0; cnt < _iterations; cnt++) {
    if (??7{true, false})
      currentVotes = ??8{counts[nextIdx − 1].idProcessed, . . .};
    if (currentVotes > ??9{counts[nextIdx − 1], . . . }.numOfMaxVotes)
      if (??10{true, false})
        ??11{counts[nextIdx − 1].maxVotes, . . . } = currentVotes;
  }
}
```

Figure 8: Contract sketch for Figure 7

4.2.4 Sketch generation

After this, SOLIDARE generates a program sketch (as in MIGRATOR). First, for each expression in the program that refers to a type affected by a transformation, that expression is replaced with a hole whose domain is all expressions with the correct type. Furthermore, for each assignment, if the assignment is affected by the transformation, it is made optional by introducing a boolean *guard hole* around it (the intuition is that some statements may become redundant as a result of the type refactoring, so they may be optimized out during sketch completion.) The sketch for the functions in our example is shown in Figure 8. As an example, observe the third line in submitForElection, which is an assignment to type *bool*. Since the second transformation has wrapped two such fields into the struct Candidate, a hole is introduced with domain including elections[nextIdx − 1].candidates[msg.sender].candidates and

elections[nextIdx − 1].candidates[msg.sender].voted.

Finally, SOLIDARE searches for a completion of the sketch generated that is equivalent to the original program, as in MIGRATOR. A key difference is that SOLIDARE uses an optimal synthesis technique to generate a program that minimizes expected gas usage. It does this by framing sketch completion as a MAXSAT instance, using soft clauses to encode a proxy of gas usage based on how holes are filled in.

The search space for this domain is much larger than SQL programs for most instances. For our example, the search space has 4096 different completions. To solve this problem, we use a similar minimal failing subcontract-based approach along with domain knowledge to prune the search space. Using these techniques, our implementation solves this problem after only three iterations and one second of runtime.

4.3 DSL Description

In this section, I will describe the domain-specific language used to describe type refactoring transformations and its semantics in terms of how a program's structure and type environments are affected by the DSL.

4.3.1 Type aliases

In our implementation, we have implemented a preprocessing step that interprets *type aliases*, which allow the user to specify an alternate name for a type, as a language extension to Solidity. For example, the program fragment

typedef myUint → *uint*;
myUint x;

is expanded to the following before being passed to the Solidity compiler:

/* **typedef** myUint → *uint*; */
/* myUint → */ *uint* x;

In addition, information about which fields have been modified is emitted for use by SOLIDARE.

Type aliases are particularly useful here because they can be used to specify certain fields to be excluded from a transformation. For example, in the following program:

struct Person {
  uint age;
  bool isTA;
  . . . }
struct Class {
  uint numStudents;
  bool hasTA; }

if we wish to perform the transformation TAProps $\leftarrow$ **Wrap**(*uint*, *bool*), we can use the type alias feature to exclude the fields of Class and perform the transformation TAProps $\leftarrow$ **Wrap**(ageType, isTAType) instead:

typedef ageType $\rightarrow$ *uint*;
typedef isTAType $\rightarrow$ *bool*;
struct Person {
  ageType age;
  isTAType isTA;
  ... }
struct Class {
  uint numStudents;
  bool hasTA; }

In addition, since our DSL requires the fields involved in a **Wrap** to have distinct types, type aliases can be used to accomplish this when necessary.

4.3.2 Syntax

Our DSL contains five types of transformations:

Wrap and Unwrap. The statement $S \leftarrow$ **Wrap**$(\tau_1, \ldots, \tau_n)$ creates a new structure S containing fields with types $\tau_1, \ldots, \tau_n$. **Unwrap**(S) is the inverse of **Wrap**, removing structure S, replacing variables with the fields contained.

Reorder. **Reorder**(S, i, j) swaps the ith and jth fields of S.

Split and Merge. The statement $S_1, S_2 \leftarrow$ **Split**(S, i) splits the structure S into two structures S_1, S_2, with S_1 containing the first i fields and S_2 the rest. Conversely, $S \leftarrow$ **Merge**(S_1, S_2) merges the fields in S_1 and S_2 into a new structure S.

Both **Split** and **Merge** are just syntactic sugar for a combination of **Wrap** and **Unwrap**, but they are included for convenience. In particular, if S has fields of type $\tau_1, \ldots, \tau_n$, the statement $(S_1, S_2) =$ **Split**(S, i) is equivalent to

$$\textbf{Unwrap}(S); S_1 \leftarrow \textbf{Wrap}(\tau_1, \ldots, \tau_i); S_2 \leftarrow \textbf{Wrap}(\tau_{i+1}, \ldots, \tau_n).$$

Similarly, if S_1 has fields $\tau_1, \ldots, \tau_n$ and S_2 has fields $\tau'_1, \ldots, \tau'_m$, then the statement $S \leftarrow$ **Merge**(S_1, S_2) is equivalent to

$$\textbf{Unwrap}(S_1); \textbf{Unwrap}(S_2); S \leftarrow \textbf{Wrap}(\tau_1, \ldots, \tau_n, \tau'_1, \ldots, \tau'_m).$$

4.4 DSL Semantics

In this section, we describe the semantics of our DSL on the structure definitions Σ and variable types Γ. We introduce the following notation:

$$\frac{\neg\,\mathrm{HasFieldSeq}(\mathcal{F}, \xi(\mathcal{W}))}{S \leftarrow \mathbf{Wrap}(\mathcal{W}) \vdash \mathcal{F} \hookrightarrow \mathcal{F}}\ \text{(Wrap1-Fld)} \qquad \frac{\neg\,\mathrm{HasField}(\mathcal{F}, \xi(S))}{\mathbf{Unwrap}(S) \vdash \mathcal{F} \hookrightarrow \mathcal{F}}\ \text{(Unwrap1-Fld)}$$

$$\frac{\mathrm{HasFieldSeq}(\mathcal{F}, \xi(\mathcal{W}))}{S \leftarrow \mathbf{Wrap}(\mathcal{W}) \vdash \mathcal{F} \hookrightarrow \mathrm{Replace}(\mathcal{F}, \xi(\mathcal{W}), \xi(S)}\ \text{(Wrap2-Fld)}$$

$$\frac{\mathrm{HasField}(\mathcal{F}, \xi(S)) \quad \mathcal{F}' = \mathrm{Fields}(S)}{\mathbf{Unwrap}(S) \vdash \mathcal{F} \hookrightarrow \mathrm{Replace}(\mathcal{F}, \xi(S), \xi(\mathcal{F}'))}\ \text{(Unwrap2-Fld)}$$

$$\frac{\begin{array}{c}\Sigma = \{S_1 \mapsto \mathcal{F}_1, \ldots, S_1 \mapsto \mathcal{F}_n\} \\ \forall i.\ S \leftarrow \mathbf{Wrap}(\mathcal{W}) \vdash \mathcal{F}_i \hookrightarrow \mathcal{F}_i'\end{array}}{S \leftarrow \mathbf{Wrap}(\mathcal{W}) \vdash \Sigma \hookrightarrow \{S \mapsto \mathcal{W}, S_1 \mapsto \mathcal{F}_1, \ldots, S_n \mapsto \mathcal{F}_n\}}\ \text{(Wrap-S)}$$

$$\frac{\begin{array}{c}\Sigma = \{S_1 \mapsto \mathcal{F}_1, \ldots, S_1 \mapsto \mathcal{F}_n\} \\ \forall i \in [1,n] - \{k\}.\ S \leftarrow \mathbf{Unwrap}(S_k) \vdash \mathcal{F}_i \hookrightarrow \mathcal{F}_i'\end{array}}{\mathbf{Unwrap}(S_k) \vdash \Sigma \hookrightarrow \{S_i \mapsto \mathcal{F}_i' \mid i \in [i,n] - \{k\}\}}\ \text{(Unwrap-S)}$$

$$\frac{\mathcal{F} = \Sigma(S) \quad \mathcal{F}' = \mathrm{Swap}(\mathcal{F}, i, j)}{\mathbf{Reorder}(S, i, j) \vdash \Sigma \hookrightarrow \Sigma[S \mapsto \mathcal{F}']}\ \text{(Reorder-S)}$$

$$\frac{\mathbf{T}_1 \vdash \Sigma \hookrightarrow \Sigma' \quad \mathbf{T}_2 \vdash \Sigma' \hookrightarrow \Sigma''}{\mathbf{T}_1; \mathbf{T}_2 \vdash \Sigma \hookrightarrow \Sigma''}\ \text{(Seq-S)}$$

Figure 9: Inference rules for structure environments

- The symbol $\mathcal{F}$ represents a field sequence, which is a sequence of types contained in a structure environment. The symbol $\mathcal{W}$ is an alternate representation for a field sequence as specified in a **Wrap** command.
- $\mathrm{HasField}(\mathcal{F}, \tau)$ is true if field sequence $\mathcal{F}$ contains a field of type τ.
- $\mathrm{HasFieldSeq}(\mathcal{F}, \mathcal{F}')$ is true if field sequence $\mathcal{F}$ contains $\mathcal{F}'$ as a *consecutive* subsequence.
- $\mathrm{Fields}(S)$ returns the field sequence consisting of all fields of structure S.
- $\mathrm{Replace}(\mathcal{F}, \mathcal{F}_1, \mathcal{F}_2)$ returns a field sequence by replacing every *consecutive* occurrence of $\mathcal{F}_1$ with $\mathcal{F}_2$ in $\mathcal{F}$.

4.4.1 Semantics over structure definitions

The semantics of our DSL on structure definitions are summarized in Figure 9. We use judgments of the form $\mathbf{T} \vdash \Sigma \hookrightarrow \Sigma'$ to mean that the DSL program $\mathbf{T}$ transforms the structure environment Σ to the new structure environment Σ'.

Semantics of Wrap. For $S \leftarrow \mathbf{Wrap}(\mathcal{F})$, we first introduce a new struct S (Wrap-S); then we substitute all occurrences of those type expressions involving those fields with S (Wrap2-Fld). Structures that do not contain $\mathcal{F}$ are unchanged. Recall that the

notation $\xi(\mathcal{F})$ refers to any sequence of types such that they consist of the sequence $\mathcal{F}$ substituted into some type expression ξ.

Example 1. Consider the following structure definition:

typedef intX → *uint*; **typedef** intY → *uint*;
struct Items {
  mapping(*uint* ⇒ *address*) owner;
  mapping(*uint* ⇒ intX) x;
  mapping(*uint* ⇒ intY) y; }

This is mapped by the transformation Point ← **Wrap**(intX, intY) as follows:

struct Point {intX x; intX y; }
struct Items {
  mapping(*uint* ⇒ *address*) owner;
  mapping(*uint* ⇒ Point) p; }

First, Wrap-S introduces a new structure Point; then, the two mappings in Items are merged into a single field, according to rule Wrap2-Fld. This is because Items contains two consecutive fields of type ξ(intX) and ξ(intY), where ξ = **mapping**(*uint* ⇒ ?), so it is replaced with ξ(Point).

Semantics of Unwrap. For **Unwrap**(S), we first remove the structure S from the structure environment (Unwrap-S). Next, as with **Wrap**, we substitute occurrences of S with the corresponding fields (Unwrap2-Fld).

Example 2. Now consider the following structure definition:

struct Point {intX x; intY y; }
struct Square {Point start; *uint* len; }

The transformation **Unwrap**(Point) generates the following new structure definition:

struct Square {intX x; intY y; }

Reorder and composition. The semantics of **Reorder**(S, i, j) are simple; the corresponding fields in S are swapped, with no other effects.

Finally, the rule Seq-S tells us how to compose two DSL statements, which is done in the expected way.

4.4.2 Semantics over variable types

The semantics of our DSL on variable types are summarized in Figure 10. We use judgments of the form $\mathbf{T} \vdash \Gamma \hookrightarrow \Gamma'$ to mean that the DSL program $\mathbf{T}$ transforms the typing environment Γ to the new typing environment Γ'.

$$\frac{\Gamma' = \Gamma[v \mapsto \xi(S) \mid \Gamma(v) = \xi(\tau_1, \ldots, \tau_n)]}{S \leftarrow \textbf{Wrap}(\tau_1, \ldots, \tau_n) \vdash \Gamma \hookrightarrow \Gamma'} \text{ (Wrap-T)}$$

$$\frac{\begin{array}{c}\text{Fields}(S) = (\tau_1, \ldots, \tau_n) \quad \Gamma' = \Gamma[v \to \bot \mid \Gamma(v) = \xi(S)] \\ \Gamma'' = \Gamma'[v_i \mapsto \xi(\tau_i) \mid \Gamma(v) = \xi(S), i \in [1, n], v_i \text{ fresh}]\end{array}}{\textbf{Unwrap}(S) \vdash \Gamma \hookrightarrow \Gamma''} \text{ (Unwrap-T)}$$

$$\frac{}{\textbf{Reorder}(S, i, j) \vdash \Gamma \hookrightarrow \Gamma} \text{ (Reorder-T)} \quad \frac{\mathbf{T}_1 \vdash \Gamma \hookrightarrow \Gamma' \quad \mathbf{T}_2 \vdash \Gamma' \hookrightarrow \Gamma''}{\mathbf{T}_1; \mathbf{T}_2 \vdash \Gamma \hookrightarrow \Gamma''} \text{ (Seq-T)}$$

Figure 10: Inference rules for type environments

Semantics of Wrap The semantics of a **Wrap** statement are described in rule Wrap-T. For the statement $S \leftarrow \textbf{Wrap}(\tau_1, \ldots, \tau_n)$, all variables whose type is included as some $\xi(\tau_i)$ has its type changed to $\xi(S)$.

Example 3. Consider the refactoring program Point $\leftarrow$ **Wrap**(intX, intY) and the following type environment Γ:

$$\Gamma(\text{xs}) = \textbf{mapping}(\textit{uint} \Rightarrow \text{intX}) \quad \Gamma(\text{ys}) = \textbf{mapping}(\textit{uint} \Rightarrow \text{intY})$$

After applying this transformation, we obtain the following new type environment Γ':

$$\Gamma'(\text{xs}) = \textbf{mapping}(\textit{uint} \Rightarrow \text{Point}) \quad \Gamma'(\text{ys}) = \textbf{mapping}(\textit{uint} \Rightarrow \text{Point})$$

Observe that in this example, one of the variables (xs or ys) is likely to be redundant as we only need one of the two. If this is the case, our synthesis algorithm, since it is guaranteed to find the optimal solution, will choose to reduce the number of used variables by eliminating one if possible.

We choose to define our DSL semantics in this way because (a) we do not know whether one is actually redundant, and (b) it is unclear what to do when there are multiple variables of the same type that have been wrapped into a structure. Therefore, the semantics keep things simple, while the optimal synthesis algorithm ensures that the transformed program does not contain redundant variables.

Semantics of Unwrap Similarly, for a statement of the form **Unwrap**(S), we apply rule Unwrap-T. First we remove all variables of type $\xi(S)$ for some ξ. Then, if S has fields of type $\tau_1, \ldots, \tau_n$, for each removed variable we introduce n new fresh variables $v_1, \ldots, v_n$ of type $\xi(\tau_1), \ldots, \xi(\tau_n)$. The new variables are blockchain variables if and only if the original variables are blockchain variables.

Example 4. Consider the refactoring program **Unwrap**(Point) and the following type environment Γ:

$$\Gamma(\text{p}) = \text{Point} \quad \Gamma(\text{m}) = \textbf{mapping}(\textit{uint} \Rightarrow \text{Point})$$

where Point has definition

$$\textbf{struct}\ \text{Point}\ \{\text{intX x; intY y; }\}.$$

After applying this transformation, we obtain the following new type environment Γ':

$$\Gamma'(\mathrm{p}_1) = \text{intX} \qquad \Gamma'(\mathrm{m}_1) = \textbf{mapping}(\textit{uint} \Rightarrow \text{intX})$$
$$\Gamma'(\mathrm{p}_2) = \text{intY} \qquad \Gamma'(\mathrm{m}_2) = \textbf{mapping}(\textit{uint} \Rightarrow \text{intY})$$

4.5 Optimal Synthesis of Smart Contracts

In the previous section, I described the semantics of our DSL in terms of how it affects the structure and type environments of a program. In this section, we use a program synthesis approach to automatically generate a new program with the refactored structure and type environments that is equivalent to the original program. The notion of equivalence we use is the same as in Solis[7], which is similar to that of Migrator, and is denoted $P \simeq P'$.

Problem statement. Our goal is to synthesize an equivalent program that minimizes gas usage, but it is difficult to estimate gas usage statically. To solve this problem, we define a proxy gas metric Ψ which takes into account the number of statements and the number of used blockchain variables. In particular, for two programs P_1, P_2, we define

$$\Psi(P_1) < \Psi(P_2) \text{ iff } |V_1| < |V_2| \vee (|V_1| = |V_2| \wedge \mathrm{Stmts}(P_1) < \mathrm{Stmts}(P_2)),$$

where V_1, V_2 are the sets of blockchain variables of P_1, P_2, respectively (note that unused variables will be eliminated during synthesis). In practice, it was found that this proxy metric is effective at comparing the quality of different solutions. With this in mind, the optimal synthesis problem can be stated as follows:

Optimal synthesis problem. Let $P = (\Sigma, \Gamma, V, \mathbf{F})$ be a smart contract and $\mathbf{T}$ a type refactoring program. Let Σ', Γ' such that $\mathbf{T} \vdash \Sigma, \Gamma \hookrightarrow \Sigma', \Gamma'$. Our synthesis problem is to find a new contract $P' = (\Sigma', \Gamma', V', \mathbf{F}')$ such that $P \simeq P'$ and $\Psi(P)$ is minimized.

4.5.1 Sketch generation

First, after transforming the structure and type environments for a program, the next thing Solidare does is generate a program sketch according to a set of rules. To give a high-level overview, the basic idea is to identify all expressions that are no longer valid, or "stale". Then, we replace each expression with a hole whose domain consists of expressions that do type check according to the context of that expression.

More formally, for a program P and transformation $\mathbf{T}$, if Γ, Γ' are type environments such that Γ is the type environment of P and $\mathbf{T} \vdash \Gamma \hookrightarrow \Gamma'$, an expression e is *stale* with respect to $\mathbf{T}$, written $\mathrm{Stale}(e)$, if for some type τ

$$\Gamma \vdash e : \tau \quad \text{and} \quad \Gamma' \nvdash e : \tau.$$

$$\frac{\tau \in \mathcal{W}}{S \leftarrow \textbf{Wrap}(\mathcal{W}) \vdash \tau \triangleright \tau}\ \text{(Wrap-T1)} \qquad \frac{\tau \in \mathcal{W} \quad \xi(\tau) \neq \tau}{S \leftarrow \textbf{Wrap}(\mathcal{W}) \vdash \xi(\tau) \triangleright \xi(S)}\ \text{(Wrap-T2)}$$

$$\frac{\tau \in \text{Fields}(S)}{\textbf{Unwrap}(S) \vdash \xi(\tau) \triangleright \xi(\tau)}\ \text{(Unwrap-T1)} \qquad \frac{\text{Fields}(S) = \mathcal{F}}{\textbf{Unwrap}(S) \vdash \xi(S) \triangleright \xi(\mathcal{F})}\ \text{(Unwrap-T2)}$$

$$\frac{\begin{array}{c}\text{LValue}(e) \quad \text{Stale}(e) \quad \Gamma \vdash e : \tau \\ \mathbf{T} \vdash \tau \triangleright \tau' \quad \delta = \{e' \mid \Gamma' \vdash e' : \tau'\}\end{array}}{\mathbf{T}, \Gamma, \Gamma' \vdash e \rightsquigarrow \textbf{??}[\delta]}\ \text{(Stale1)}$$

$$\frac{\begin{array}{c}\text{LValue}(e) \quad \text{Stale}(e) \quad \Gamma \vdash e : \tau \\ \mathbf{T} \vdash \tau \triangleright (\tau'_1, \ldots, \tau'_n) \\ \forall i.\ \delta_i = \{e' \mid \Gamma \vdash e' : \tau'_i\}\end{array}}{\mathbf{T}, \Gamma, \Gamma' \vdash e \rightsquigarrow (\textbf{??}[\delta_1], \ldots, \textbf{??}[\delta_2])}\ \text{(Stale2)} \qquad \frac{\text{LValue}(e) \quad \neg\,\text{Stale}(e)}{\mathbf{T}, \Gamma, \Gamma' \vdash e \rightsquigarrow e}\ \text{(NonStale)}$$

$$\frac{\begin{array}{c}\forall i.\ \mathbf{T}, \Gamma, \Gamma' \vdash e_i \rightsquigarrow e'_i \\ \circ \text{ is an operator, function call, structure constructor}\end{array}}{\mathbf{T}, \Gamma, \Gamma' \vdash \circ(e_1, \ldots, e_n) \rightsquigarrow \circ(e'_1, \ldots, e'_n)}\ \text{(Comp)}$$

$$\frac{\delta' = \{e' \mid \mathbf{T}, \Gamma, \Gamma' \vdash e \rightsquigarrow e', e \in \text{AllDoms}(\delta)\}}{\mathbf{T}, \Gamma, \Gamma' \vdash \textbf{??}[\delta] \rightsquigarrow \textbf{??}[\delta']}\ \text{(Hole)}$$

Figure 11: Inference rules for expressions. $\text{LValue}(e)$ is true if e is a variable, field access $e'.f$, or index operation $e'[a]$, where e' is an arbitrary expression and f, a are an arbitrary field and expression.

For example, in Example 3, the expression xs[0] is a stale expression because $\Gamma \vdash$ xs[0] : intX (i.e. it has type intX), but $\Gamma' \nvdash$ xs[0] : intX because in the new type environment xs[0] has type Square. Furthermore, in Example 4, the expression p is stale because the variable no longer exists in Γ'.

Sketch generation for expressions. After identifying stale expressions, we must replace them with new expressions that type-check.

Before we can describe how sketch generation works for expressions, we first need to define how the type of an expression in the original program maps to a new type in the transformed program. We do this with a type correspondence relation, written $\tau \triangleright \tau'$, described in the first four rules of Figure 11. It is important to note that even if an expression e has type τ such that $\tau \triangleright \tau'$, the expression may (and if it is stale, will not) still type-check under Γ' and must be replaced with a new expression. Next I will describe in detail how this correspondence is defined for **Wrap** and **Unwrap** (recall that **Reorder** will never cause an expression to be stale.)

*Type correspondence for **Wrap**.* There are two cases that can occur for stale expressions as a result of $S \leftarrow \textbf{Wrap}(\tau_1, \ldots, \tau_n)$. This concerns expressions whose type τ_i is contained in the field sequence of **Wrap**. For variables of type τ_i (rule Wrap-T1), the type remains the same, as the underlying types τ_i are still valid. However, for

nested types, the situation is different. This is because blockchain variables of type τ_i have been wrapped inside the structure S. Therefore, when an expression has type $\xi(\tau_i) \neq \tau_i$ (rule Wrap-T2), the corresponding type in the refactored program is $\xi(S)$.

*Type correspondence for **Unwrap**.* Like in **Wrap**, the types of expressions with type $\tau_i \in \text{Fields}(S)$ after **Unwrap**(S) remain the same (rule Unwrap-T1). Unlike **Wrap**, this holds even if τ_i is inside a nested type $\xi(\tau_i)$. However, as shown in rule Unwrap-T2, variables of type $\xi(S)$ must be unwrapped into n different variables of type $\xi(\tau_1), \ldots, \xi(\tau_n)$, where $(\tau_1, \ldots, \tau_n) = \text{Fields}(S)$.

As mentioned at the beginning of this section, the key idea behind sketch generation is to replace each stale expression with a hole whose domain includes well-typed expressions. To formalize this notion, we introduce the following definition:

Valid replacement. Let $\mathbf{T}$ be a type refactoring for program $P = (\Sigma, \Gamma, V, \mathbf{F})$ and Σ', Γ' such that $\mathbf{T} \vdash \Sigma, \Gamma \hookrightarrow \Sigma', \Gamma'$. Then an expression e' is a *valid replacement* for an expression $e \in P$ if and only if for some τ',

$$(1)\ \mathbf{T} \vdash \tau \triangleright \tau' \quad \text{and} \quad (2)\ \Gamma' \vdash e' : \tau'.$$

Rules for sketch generation. Using this notion of valid replacements, I will now describe the sketch generation procedure for expressions. This is shown in the last 5 rules of Figure 11, where $\mathbf{T}, \Gamma, \Gamma' \vdash e \rightsquigarrow e'$ means that e should be rewritten to e' (similar to MIGRATOR.)

As previously mentioned, we replace stale expressions with a hole whose domain includes all valid replacements (Stale1). Rule Stale2 generalizes this to the case when a variable should be replaced with several new variables, as with **Unwrap**. Non-stale expressions are left alone (NonStale), which is useful when combined with the next rule: complex expressions, such as those with arithmetic operators and function calls, are rewritten recursively (Comp). Finally, when rewriting holes (as is done with multi-statement refactorings), we recursively generate holes for each expression in the domain and then combine their domains.

Sketch generation for statements. The rules for sketch generation for statements are summarized in Figure 12. The basic idea is to replace each assignment statement s that contains a stale expression with a new statement **if** (**??**$\{\texttt{true}, \texttt{false}\}$) s', where s' is obtained by replacing the stale expressions in s according to the previous section. The reason why we introduce guards is that some statements can become redundant after refactoring; in order for the resulting program to be optimal, we must eliminate them during sketch completion.

The first three rules describe how a sketch is generated for assignments $l := e$. If neither l nor e is stale, the assignment is unchanged (Assign1). Otherwise, the assignment is stale, and we replace the stale sides according to the previous section. If the resulting types are still not tuples, we generate a conditional assignment (Assign2); otherwise, we lift the assignment to generate multiple (optional) assignments for each element (Assign3).

The remaining rules (Seq, Cond, Loop) describe how to compose statements by proceeding recursively.

$$\frac{\neg\,\mathrm{HasStaleExpr}(l) \quad \neg\,\mathrm{HasStaleExpr}(e)}{\mathbf{T},\Gamma,\Gamma' \vdash l \coloneqq e \rightsquigarrow l \coloneqq e}\ (\text{Assign1}) \qquad \frac{\mathrm{HasStaleExpr}(l \vee e) \quad \mathbf{T},\Gamma,\Gamma' \vdash l \rightsquigarrow l', e \rightsquigarrow e'}{\mathbf{T},\Gamma,\Gamma' \vdash l \coloneqq e \rightsquigarrow l' \blacktriangleleft e'}\ (\text{Assign2})$$

$$\frac{\mathrm{HasStaleExpr}(l) \quad \mathbf{T},\Gamma,\Gamma' \vdash l \rightsquigarrow (l'_1,\ldots,l'_n) \quad \mathrm{HasStaleExpr}(e) \quad \mathbf{T},\Gamma,\Gamma' \vdash e \rightsquigarrow (e'_1,\ldots,e'_n)}{\mathbf{T},\Gamma,\Gamma' \vdash l \coloneqq e \rightsquigarrow l'_1 \blacktriangleleft e'_1;\ldots;l'_n \blacktriangleleft e'_n}\ (\text{Assign3})$$

$$\frac{\mathbf{T},\Gamma,\Gamma' \vdash s_1 \rightsquigarrow s'_1, s_2 \rightsquigarrow s'_2}{\mathbf{T},\Gamma,\Gamma' \vdash s_1; s_2 \rightsquigarrow s'_1; s'_2}\ (\text{Seq})$$

$$\frac{\mathbf{T},\Gamma,\Gamma' \vdash e \rightsquigarrow e', s_1 \rightsquigarrow s'_1, s_2 \rightsquigarrow s'_2}{\mathbf{T},\Gamma,\Gamma' \vdash \textbf{if}\ (e)\ s_1\ \textbf{else}\ s_2 \rightsquigarrow \textbf{if}\ (e')\ s'_1\ \textbf{else}\ s'_2}\ (\text{Cond})$$

$$\frac{\mathbf{T},\Gamma,\Gamma' \vdash e \rightsquigarrow e', s_1 \rightsquigarrow s'_1}{\mathbf{T},\Gamma,\Gamma' \vdash \textbf{while}\ (e)\ s_1 \rightsquigarrow \textbf{while}\ (e')\ s'_1}\ (\text{Loop})$$

Figure 12: Inference rules for statements. The notation $l \blacktriangleleft e$ is shorthand for the optional assignment $\textbf{if}\ (\texttt{??}\{\texttt{true}, \texttt{false}\})\ l \coloneqq e$.

Example 5. Consider the refactoring from Example 4 and the statement $\mathrm{m}[0] \coloneqq \mathrm{Point}(\mathrm{x}, \mathrm{y})$ with $\Gamma(\mathrm{x}) = \mathrm{intX}, \Gamma(\mathrm{y}) = \mathrm{intY}$. We generate the following sketch:

$$\textbf{if}\ (\texttt{??}_1\{\texttt{true}, \texttt{false}\})\ \texttt{??}_2\{\mathrm{m}_1[0], \ldots\} \coloneqq \texttt{??}_3\{x, \ldots\}$$
$$\textbf{if}\ (\texttt{??}_4\{\texttt{true}, \texttt{false}\})\ \texttt{??}_5\{\mathrm{m}_2[0], \ldots\} \coloneqq \texttt{??}_6\{y, \ldots\}$$

Multi-statement refactorings. We apply the rules in the previous sections repeatedly, as described using the following inference rules:

$$\frac{\mathbf{T} \vdash \Gamma \hookrightarrow \Gamma' \quad \mathbf{T},\Gamma,\Gamma' \vdash s \rightsquigarrow s'}{\mathbf{T},\Gamma,\Gamma' \vdash s \rightsquigarrow^* s'}\ (\text{Base})$$

$$\frac{\mathbf{T}_1,\Gamma,\Gamma' \vdash s \rightsquigarrow^* s' \quad \mathbf{T}_2,\Gamma',\Gamma'' \vdash s' \rightsquigarrow^* s''}{\mathbf{T}_1;\mathbf{T}_2,\Gamma,\Gamma'' \vdash s \rightsquigarrow^* s''}\ (\text{Ind})$$

The idea is to first generate a sketch for the first *atomic transformation* (**Wrap**, **Unwrap**, etc.); then, apply the next atomic transformation to the resulting sketch, and so on.

4.5.2 Sketch completion

The basic sketch completion approach of Solidare is essentially the same as Migrator, but with a few modifications to make it more suitable for this new domain. Like in Migrator, we reduce the problem of filling in each hole with an element of its domain to SAT.

Hard constraints. First, like in MIGRATOR, we must ensure that each hole is filled in with exactly one candidate. This is encoded using the following hard constraints:

$$\bigwedge_{\texttt{??}_i \in \mathrm{Hole}(\Omega)} \oplus(b_i^1, \ldots, b_i^{n_i}).$$

Additionally, since programs contain many instances of the same expression, we can exploit this symmetry by requiring that those holes are instantiated in the same way:

$$\bigwedge_{e} \bigvee_{e'} \bigwedge_{\texttt{??}_i | e \rightsquigarrow^* \texttt{??}_i} b_i^{\mathrm{Index}(e', \texttt{??}_i)},$$

where $\mathrm{Index}(e', \texttt{??}_i)$ is the index of expression e' in hole i. Writing this with quantifiers makes it a bit easier to understand (where $\mathrm{Holes}(e) = \{\texttt{??}_i \mid e \rightsquigarrow^* \texttt{??}_i\}$):

$$\forall e \in P.\ \exists e'.\ \forall \texttt{??}_i \in \mathrm{Holes}(e).\ \texttt{??}_i \mapsto e'.$$

Example 6. Consider the transformation from Example 3 and the statements

$$\begin{aligned} &\mathrm{z} \coloneqq \mathrm{xs}[0]; \quad (\Gamma(\mathrm{z}) = \mathrm{intX}) \\ &\mathrm{xs}[1] \coloneqq \mathrm{xs}[1] + \mathrm{z}; \end{aligned}$$

According to our sketch generation rules (using f_1 to represent the freshly generated field), the corresponding sketch is (where $\delta_1 = \{\mathrm{z}.f_1, \mathrm{xs}[0].f_1\}, \delta_2 = \{\mathrm{z}.f_1, \mathrm{xs}[0].f_1, \mathrm{xs}[1].f_1\}$):

$$\begin{aligned} &\textbf{if } (\texttt{??}_1\{\texttt{true}, \texttt{false}\})\ \texttt{??}_2[\delta_1] \coloneqq \texttt{??}_3[\delta_1] \\ &\textbf{if } (\texttt{??}_4\{\texttt{true}, \texttt{false}\})\ \texttt{??}_5[\delta_2] \coloneqq \texttt{??}_6[\delta_2] + \texttt{??}_7[\delta_2] \end{aligned}$$

Here, $\texttt{??}_5$ and $\texttt{??}_6$ both correspond to the source expression $\mathrm{xs}[1]$. Therefore, we add the following constraint (we could also do so for source expression z):

$$(b_5^1 \wedge b_6^1) \vee (b_5^2 \wedge b_6^2) \vee (b_5^3 \wedge b_6^3).$$

In other words, both of these holes must be instantiated with either $\mathrm{z}.f_1$, $\mathrm{xs}[0].f_1$, or $\mathrm{xs}[1].f_1$.

Soft constraints. However, we also have the additional constraint that our proxy gas metric Ψ is minimized. Therefore, we use soft constraints to guide the search towards instantiations with less gas usage.

Minimizing blockchain variables. First, I will consider how to minimize the number of used blockchain variables. Let $\mathrm{Guards}(\texttt{??}_i)$ denote the set of all guard holes surrounding hole i. For each blockchain variable v, if all guard holes for hole i are assigned to true (which always has index 1), we prefer solutions that do not assign $\texttt{??}_i$ to an expression that contains v. This is encoded using the following soft constraint, where n is the number of statements in the sketch:

$$\bigwedge_{v \in V} \Big(\bigwedge_{\texttt{??}_i \in \mathrm{Holes}(\Omega)} \Big(\bigwedge_{\texttt{??}_j \in \mathrm{Guards}(\texttt{??}_i)} b_k^i \rightarrow \Big(\bigwedge_{e_j \in \mathrm{Dom}(\texttt{??}_i)} (v \in e_j) \rightarrow \neg b_i^j \Big) \Big), n \Big).$$

Example 7. Consider the sketch in Example 6 and assume z is a blockchain variable. In the sketch, z occurs in holes $??_2$ and $??_3$ (both guarded by $??_1$) as well as $??_5, ??_6, ??_7$ (guarded by $??_4$). This translates to the following soft constraint:

$$\big(b_1^1 \to (\neg b_2^1 \wedge \neg b_3^1)\big) \wedge \big(b_4^1 \to (\neg b_5^1 \wedge \neg b_6^1 \wedge \neg b_7^1)\big).$$

This says that (recall that z.f_1 is the first element in the domain for the holes in which z occurs): if the guards are enabled (b_1^1 and b_4^1), they should not be instantiated with z.f_1.

Blocking clause generation. Similar to MIGRATOR, we want to generate a smart blocking clause the prevents a large number of incorrect completions. First, we introduce the notion of a *minimal failing subcontract*:

Minimal failing subcontract. Let the notation $P \downarrow \mathbf{F}$ denote the same contract as P but only containing functions in the set $\mathbf{F}$. Given a source contract $P = (\Sigma, \Gamma, V, \mathbf{F})$ and candidate refactored contract $P' = (\Sigma', \Gamma', V', \mathbf{F}')$, a *minimal failing subcontract* is $P^* = (\Sigma', \Gamma', V', \mathbf{F}^*)$ such that

$$(1)\ \mathbf{F}^* \subseteq \mathbf{F}', \quad (2)\ P \downarrow \mathbf{F}^* \not\simeq P^*, \quad (3)\ \forall \hat{\mathbf{F}} \subset \mathbf{F}^*.\ P \downarrow \hat{\mathbf{F}} \simeq P' \downarrow \hat{\mathbf{F}}.$$

In other words, given an incorrect program P', a minimal failing subcontract P^* is one that (1) contains a subset of the functions in P', (2) the behavior of P^* is not equivalent to the original contract, and (3) P^* is minimal in the sense that removing any other function definition causes the resulting contract to be equivalent to the original contract with respect to the functions defined.

Example 8. Consider the following smart contract P that stores a point with coordinates:

```
contract SimplePoint {
  uint public x = 0; uint public y = 0;
  function set(uint _x, uint _y) public {x = _x; y = _y;}
  function getX() public returns (uint) {return x;}
  function getY() public returns (uint) {return y;} }
```

and another smart contract P^* as follows:

```
contract SimplePoint {
  uint public x = 0; uint public y = 0;
  function set(uint _x, uint _y) public {x = _x; y = _y;}
  function getX() public returns (uint) {return y;} }
```

Here, P^* is a minimal failing subcontract of P: (1) P^* contains a subset of the functions in P: set and getX; (2) the behavior of P^* is different from P because executing set$(1, 2)$; getX$()$ returns 1 on P but 2 on P^*; (3) P^* is minimal because removing either function set or getX results in a subcontract that is equivalent to the corresponding part of P. As in MIGRATOR, any refactoring that agrees with P^* on the holes used in P^* cannot be correct, so we can block only those holes used in P^*.

Example 9. Let us continue with Example 8. Now consider a sketch S that is generated from the original program P (where $\delta_1 = \{\_\text{x}, \_\text{y}\}, \delta_2 = \{\text{x}, \text{y}\}$):

contract SimplePoint {
uint **public** x = 0; *uint* **public** y = 0;
function set(*uint* _x, *uint* _y) **public** {x = **??**$_1[\delta_1]$; y = **??**$_2[\delta_1]$; }
function getX() **public returns** (*uint*) {**return** **??**$_3[\delta_2]$; }
function getY() **public returns** (*uint*) {**return** **??**$_4[\delta_2]$; } }

Since getY is not a part of the minimal failing subcontract from Example 8, the assignment to **??**$_4$ is irrelevant, so we can prune all sketch completions where **??**$_1 \mapsto$ _x, **??**$_2 \mapsto$ _y, **??**$_3 \mapsto$ _y.

Generating blocking clauses from minimal failing subcontracts. To encode this as a blocking clause, given a minimal failing subcontract $P^* = \Omega^*[M^*]$ with sketch and model (Ω^*, M^*) of candidate refactored contract $P' = \Omega'[M']$, where $M^*(\texttt{??}_i)$ is the index of the assignment to hole i and $M^* \vdash \texttt{??}_i \mapsto e$ means that M^* maps hole i to expression e, we add the following constraint, where k_i is the index of the assignment to hole i:

$$\phi = \neg\left(\bigwedge_{\texttt{??}_i \in \text{Holes}(\Omega^*)} (\forall \texttt{??}_k \in \text{Guards}(\texttt{??}_i).\ M^* \vdash \texttt{??}_k \mapsto \top) \rightarrow b_i^{M^*(\texttt{??}_i)}\right).$$

In particular, if we say that a hole **??**$_i$ is *enabled* by model M^* if M^* assigns all of **??**$_i$'s guards to true, any model M'' that agrees with M' on the assignments to enabled holes is also guaranteed to yield an incorrect completion. Therefore, when generating a blocking clause we can safely disregard all assignments to variables that are either not in the domain of M' or disabled by M'.

4.6 Implementation

In this section, I will discuss some optimizations we employed in our implementation over the basic synthesis algorithm as presented.

4.7 Sketch Generation

Deterministic split rules. Instead of implementing **Split** as a sequence of **Unwrap** and **Wrap** statements, due to the specific structure of **Split**, we were able to implement it as a fully deterministic transformation. In particular, if a structure S is split into two structures S_1, S_2, then for each variable x of type $\xi(S)$ we introduce two variables of type $\xi(S_1), \xi(S_2)$. Then, for every field access of the form $e.f$ where e is an expression that contains x, we translate this to $e'.f$ where e' is $e[x_i/x]$ for $x_i \in \{x_1, x_2\}$, depending on whether the field f belongs to S_1 or S_2. The full sketch generation rules will be available in the Appendix[4].

Domain generation. As described, the sketch generation procedure specifies that the domain of holes contains all expressions of the replacement type. However, this set is very large and possibly infinite. Therefore, we restrict the domain of each hole to expressions that can be formed using subterms of the source expression along with newly-generated fields and variables. We also discard expressions whose index structure is not compatible with the source expression; for example, if the source expression is x, it is never necessary or profitable to generate ys[z] as a replacement.

4.8 Sketch Completion

In addition to the basic MaxSAT encoding, we also include two additional constraints we found to be useful in practice. First, we disallow completions where the left-hand side and right-hand side of an assignment are syntactically identical (and are sufficiently "simple"). Second, we restrict completions such that if a variable is read, it must be written to at some point in the program (recall that the order in which functions are executed is not known).

4.9 Equivalence Checking

We use the Solis[7] equivalence checking engine for Solidity, but as in Migrator, we first perform bounded testing to quickly refute most incorrect completions and only invoke the more sophisticated checker when testing does not find a counterexample.

4.10 Generating Minimal Failing Subcontracts

Recall that we use minimal failing subcontracts to prune large parts of the search space using a single counterexample. One obvious way to do so is to successively remove functions and check that it is a minimal failing subcontract. However, this approach is potentially very slow as it involves many calls to the verifier. In the case of a minimum failing input (as in Migrator) returned by exhausting testing, we know that the resulting subcontract is guaranteed to be minimal. However, for counterexamples not found through testing, the subcontract is not guaranteed to be minimal; nonetheless, we found the the number of functions is still rather small and we found that removing only the functions not used in the counterexample is still very effective in practice.

4.11 Soundness and Completeness

As shown in the appendix[4], Solidare is both sound, which means that generated programs are guaranteed to be equivalent to the original program as well as minimizing the proxy gas metric Ψ, and complete, which means that if there exists some program $P' \simeq P$, Solidare is guaranteed to find that program.

4.12 Evaluation

We have evaluated Solidare on a dataset of 20 smart contracts taken from Etherscan and a total of 39 manually-written transformations designed to reduce gas usage. See

ID	Contract	LOC	Fns	T	Sketch Time (s)	Completion Time (s)	Max Diff	Avg Diff
1	Announcement	112	7	2	0.2	0.2	23	17.5
2	Auction	964	70	1	3.7	7.7	34	34.0
3	BdpImageStorage	258	27	2	0.1	0.4	32	32.0
4	BinaryOption	916	20	1	0.4	1.4	31	31.0
5	Congress	163	9	3	1.2	1.6	66	34.7
6	CreditDAO	111	14	2	0.4	0.4	54	50.0
7	CryptoTask	255	17	3	0.3	0.3	12	8.3
8	DAOG2X	319	19	3	0.7	1.9	24	23.0
9	EMPresale	306	30	3	0.7	551.1	57	38.0
10	EthLottery	132	6	2	0.3	0.2	22	21.5
11	EtherRacing	250	20	2	1.2	6.2	32	32.0
12	FTICrowdsale	553	17	1	0.1	0.3	9	9.0
13	JanKenPon	510	40	1	17.0	2.7	47	47.0
14	Kingdom	189	13	3	0.6	3.5	64	44.7
15	Oryza	152	7	2	1.0	1.4	21	20.0
16	PollManager	473	12	2	3.0	3.1	17	17.0
17	Slaughter3D	287	26	1	0.7	2.2	22	22.0
18	SplitStealContract	465	28	2	5.0	3.9	24	23.5
19	TwoXJackpot	222	15	1	0.4	1.3	14	14.0
20	moduleToken	392	21	2	0.6	0.8	18	18.0

Table 3: Statistics about benchmarks and SOLIDARE's running time.

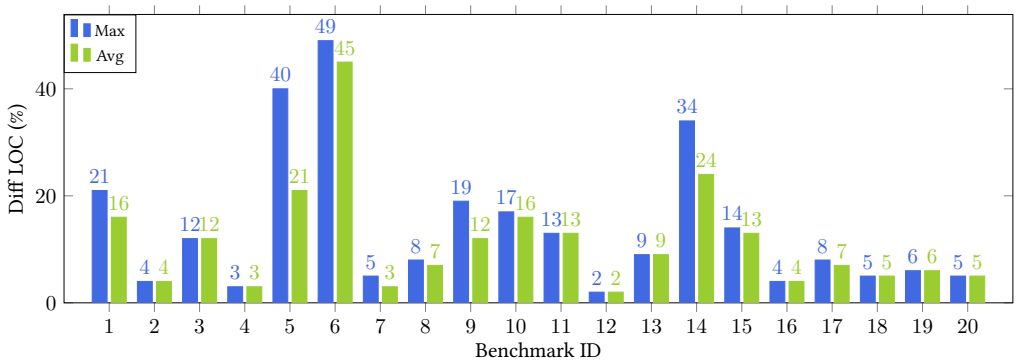

Figure 13: Diff size as percentage of the lines of code in original contracts.

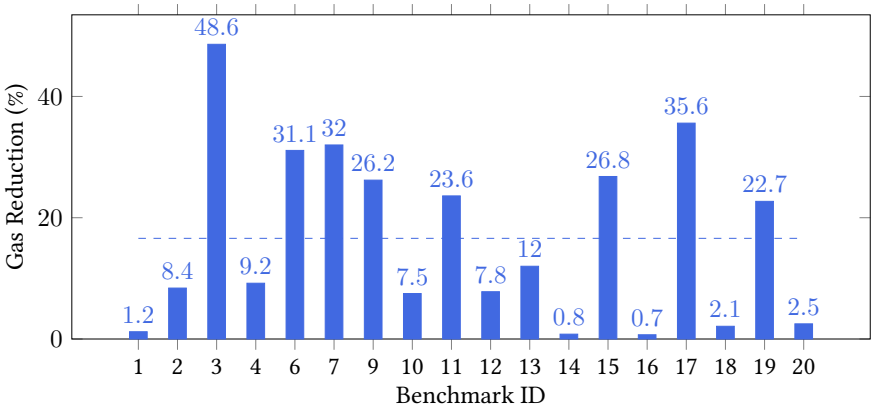

Figure 14: Gas reduction in benchmarks.

the original paper[4] for details on how these benchmarks were selected and experimental setup.

The results are summarized in Table 3. We can see that Solidare is able to synthesize semantically equivalent programs for all 39 transformations, and its average running time is 31.4 seconds. Note that due to the presence of an outlier (EMPresale), this average is higher than the median, which is 2.5 seconds. In the case of EMPresale, this is because the generated sketch is very large (search space of $\approx 8.8 \times 10^{12}$) and requires 7756 sketch completions before finding the correct one. See [4] for a more detailed outlier analysis.

Furthermore, as can be seen in Figure 13, many transformations require changing a significant portion of the source contract, up to 49% in one case. This indicates that Solidare is a useful tool for automatically performing data structure refactoring with minimal effort.

Gas reduction. Since the main motivation of this technique is to reduce gas usage of smart contracts, we have measured gas usage for both the original and refactored contracts according to a representative workload based on statistics on how frequently each function is invoked. The results are shown in Figure 14.

Effectiveness of optimal synthesis and minimal failing subcontracts. In order to evaluate the effectiveness of our proposed approach, we have compared Solidare against three variants that do not use some techniques:

- Solidare-NoMFS is a variant that does not utilize minimal failing subcontracts to generate blocking clauses.
- Solidare-NoSoft is a variant that does not use soft clauses to ensure optimality. Instead, it uses enumerative checking (but still using minimal failing subcontracts) to find the best solution.

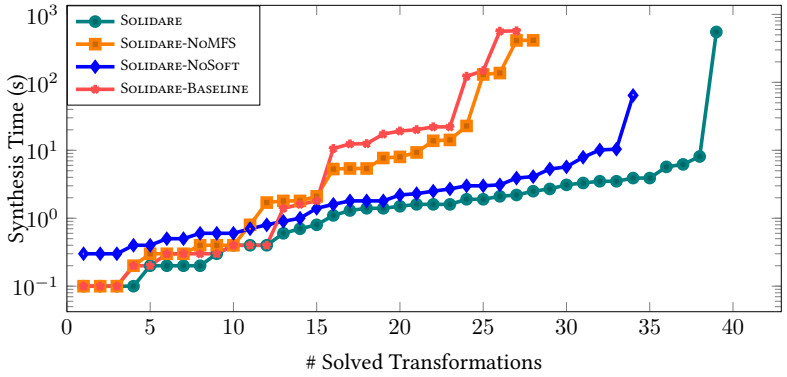

Figure 15: Comparing Solidare against baselines. y-axis is on log-scale.

- Solidare-Baseline is a variant that does not use soft clauses to ensure optimality nor minimal failing subcontracts.

As can be seen in Figure 15, our proposed sketch completion algorithm significantly outperforms the three baselines, which fail to solve 13% (NoSoft), 28% (NoMFS), and 31% (Baseline) of the benchmarks.

5 Conclusion

In this paper, I have presented two tools for automating data structure refactoring.

First, Migrator allows programmers to automatically refactor database programs based on a destination schema. In particular, it does not require any other specification or input from the programmer and is completely automated. In addition, it is both sound and relatively complete, so the programmer has a guarantee that the resulting program has the same semantics as the original program.

Second, Solidare allows programmers to perform data structure refactoring on smart contracts using a simple domain-specific language. Although we require this extra specification from the programmer, we decided that the flexibility afforded allows the programmer to experiment with different transformations to find one that achieves optimal gas efficiency. Like Migrator, it is both sound and complete.

Future work. Although both of these tools are already very useful in their current form, there is still potential work to be done in order to make them even more useful.

For Migrator, in addition to extending the notion of value correspondence to allow for more expressive transformations, it would be useful if it could automatically generate a procedure that migrates an existing database to the new schema. Furthermore, in addition to allowing the programmer to specify the target schema, a potential improvement would be if Migrator could automatically generate a program which is optimal with regard to some criterion that encodes real-world performance.

For Solidare, one straightforward improvement would be to refine the proxy gas model to more accurately predict real-world gas usage. Another useful improvement would be to allow the programmer to specify only the target data structure as in Migrator. Finally, like Migrator, it would be useful if Solidare could automatically generate transformations to optimize gas usage without a specification from the programmer.

6 Acknowledgements

I would first like to thank Dr. Işıl Dillig for her support throughout my entire undergraduate career. Her expert guidance has greatly impacted my research experience, and there is no way I could have accomplished this work without her.

I would also like to thank Dr. Yuepeng Wang for working with me in both of these projects. His dedication and expertise were instrumental in bringing them to fruition, and he was always patient and helpful during our discussions.

Finally, I would like to thank Rushi Shah for his work on Migrator as well as Maruth Goyal, Yanju Chen, and Dr. Yu Feng for their work on Solidare. In particular, Maruth's enthusiasm for research was greatly motivating.

References

[1] Elvira Albert, Jesús Correas, Pablo Gordillo, Guillermo Román-Díez, and Albert Rubio. Gasol: Gas analysis and optimization for ethereum smart contracts. In Armin Biere and David Parker, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 118–125, Cham, 2020. Springer International Publishing.

[2] 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, page 422–436, New York, NY, USA, 2017. Association for Computing Machinery.

[3] 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*, PLDI '15, page 229–239, New York, NY, USA, 2015. Association for Computing Machinery.

[4] Maruth Goyal, James Dong, Yanju Chen, Yuepeng Wang, Yu Feng, and Isil Dillig. Synthesis-powered optimization of smart contracts via data type refactoring. Under review, 2021.

[5] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, page 317–330, New York, NY, USA, 2011. Association for Computing Machinery.

[6] Daniel Le Berre and Anne Parrain. The SAT4J library, release 2.2, system description. *Journal on Satisfiability Boolean Modeling and Computation*, 7:59–64, 07 2010.

[7] Ben Mariano, Yanju Chen, Yu Feng, Shuvendu Lahiri, and Isil Dillig. Demystifying loops in smart contracts. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE'20)*. IEEE/ACM, IEEE/ACM, September 2020.

[8] Renée J. Miller, Laura M. Haas, and Mauricio A. Hernández. Schema mapping as query discovery. In *Proceedings of the 26th International Conference on Very Large Data Bases*, VLDB '00, page 77–88, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[9] Paul Monica. Move over, bitcoin. ethereum is at an all-time high. *CNN Business*, 2021.

[10] Armando Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, University of California at Berkeley, USA, 2008. AAI3353225.

[11] Yuepeng Wang, Isil Dillig, Shuvendu K. Lahiri, and William R. Cook. Verifying equivalence of database-driven applications. *Proc. ACM Program. Lang.*, 2(POPL), December 2017.

[12] Yuepeng Wang, James Dong, Rushi Shah, and Isil Dillig. Synthesizing database programs for schema refactoring. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 286–300, New York, NY, USA, 2019. Association for Computing Machinery.

[13] Yuepeng Wang, James Dong, Rushi Shah, and Isil Dillig. Synthesizing database programs for schema refactoring. https://arxiv.org/abs/1904.05498, 2019. Extended version.

[14] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. https://ethereum.github.io/yellowpaper/paper.pdf, 2021. Accessed 2021-05-03; last updated 2021-02-14.