Synthesis-Powered Optimization of Smart Contracts via Data Type Refactoring

YANJU CHEN*, University of California, Santa Barbara, USA
YUEPENG WANG*, Simon Fraser University, Canada
MARUTH GOYAL, University of Texas at Austin, USA
JAMES DONG, Stanford University, USA
YU FENG, University of California, Santa Barbara, USA
ISIL DILLIG, University of Texas at Austin, USA

Since executing a smart contract on the Ethereum blockchain costs money (measured in *gas*), smart contract developers spend significant effort in reducing gas usage. In this paper, we propose a new technique for reducing the gas usage of smart contracts by changing the underlying data layout. Given a smart contract $\mathcal{P}$ and a type-level transformation, our method automatically synthesizes a new contract $\mathcal{P}'$ that is functionally equivalent to $\mathcal{P}$. Our approach provides a convenient DSL for expressing data type refactorings and employs program synthesis to generate the new version of the contract. We have implemented our approach in a tool called Solidare and demonstrate its capabilities on real-world smart contracts from Etherscan and GasStation. In particular, we show that our approach is effective at automating the desired data layout transformation and that it is useful for reducing gas usage of smart contracts that use rich data structures.

CCS Concepts: • **Software and its engineering → Automatic programming**; **Search-based software engineering**.

ACM Reference Format:
Yanju Chen, Yuepeng Wang, Maruth Goyal, James Dong, Yu Feng, and Isil Dillig. 2022. Synthesis-Powered Optimization of Smart Contracts via Data Type Refactoring. *Proc. ACM Program. Lang.* OOPSLA, 1, Article 1 (January 2022), 29 pages. https://doi.org/10.xxxx/xxxxxxx.xxxxxxx

1 INTRODUCTION

Smart contracts are programs that run on the blockchain and programmatically enforce contracts between multiple parties. Since many blockchains (e.g., Ethereum) need *miners* to perform computation and pay miners a fee in return, executing a smart contract requires an amount of money (measured in *gas*) that is proportional to its computational cost. Thus, smart contract developers typically invest significant effort in optimizing their code and making it as gas-efficient as possible.

Motivated by this problem, there has been recent interest in optimization techniques for reducing the gas usage of smart contracts, including bytecode superoptimization [Albert et al. 2020a,b; Nagele and Schett 2020] and anti-pattern detection [Chen et al. 2017, 2018]. However, reducing the gas

*Both authors contributed equally to the paper.

Authors' addresses: Yanju Chen, University of California, Santa Barbara, USA, yanju@cs.ucsb.edu; Yuepeng Wang, Simon Fraser University, Canada, yuepeng@sfu.ca; Maruth Goyal, University of Texas at Austin, USA, maruth@cs.utexas.edu; James Dong, Stanford University, USA, dongj@stanford.edu; Yu Feng, University of California, Santa Barbara, USA, yufeng@cs.ucsb.edu; Isil Dillig, University of Texas at Austin, USA, isil@cs.utexas.edu.

2475-1421/2022/1-ART1 $15.00
https://doi.org/10.xxxx/xxxxxxx.xxxxxxx

```
contract CreditDAO {
    struct Election {
        address maxVotes;
        uint nextCandidateIndex;
        mapping(address => bool) candidates;
        mapping(address => bool) userHasVoted;
        mapping(uint => uint) candidateVotes;


        uint numMaxVotes;
        uint idProcessed;
    }


    uint public nextEId;
    mapping(uint => Election) public elections;

    constructor() public {
        nextEId++;
    }
    function sumbitForElection() public {
        elections[nextEId-1].nextCandidateIndex++;
        elections[nextEId-1].candidates[msg.sender] = true;
    }
    function vote(uint candidateId) public {
        elections[nextEId-1].candidateVotes[candidateId] += 1;
        elections[nextEId-1].userHasVoted[msg.sender] = true;
    }
    function finishElections(uint _iterations) public {
        uint currentVotes;
        Election election = elections[nextEId-1];
        uint nextId = election.idProcessed;

        for (uint cnt = 0; cnt < _iterations; cnt++) {
            currentVotes = election.candidateVotes[nextId];
            if (currentVotes > election.numMaxVotes) {
                election.numMaxVotes = currentVotes;
            }
            nextId++;
        }
        election.idProcessed = nextId;
    }
}
```

```
contract CreditDAO {
    struct Election {
        address maxVotes;
        uint nextCandidateIndex;
        mapping(address => Participant) userMap;

        mapping(uint => uint) candidateVotes;
    }
    struct Count {
        uint numMaxVotes;
        uint idProcessed;
    }
    struct Participant {
        bool isCandidate;
        bool hasVoted;
    }
    uint public nextEId;
    mapping(uint => Election) public elections;
    mapping(uint => Count) public counts;
    constructor() public {
        nextEId++;
    }
    function sumbitForElection() public {
        elections[nextEId-1].nextCandidateIndex++;
        elections[nextEId-1].userMap[msg.sender].isCandidate = true;
    }
    function vote(uint candidateId) public {
        elections[nextEId-1].candidateVotes[candidateId] += 1;
        elections[nextEId-1].userMap[msg.sender].hasVoted = true;
    }
    function finishElections(uint _iterations) public {
        uint currentVotes;
        Election election = elections[nextEId-1];
        Count count = counts[nextEId-1];
        uint nextId = count.idProcessed;
        for (uint cnt = 0; cnt < _iterations; cnt++) {
            currentVotes = election.candidateVotes[nextId];
            if (currentVotes > count.numMaxVotes) {
                count.numMaxVotes = currentVotes;
            }
            nextId++;
        }
        count.idProcessed = nextId;
    }
}
```

Fig. 1. A motivating example to demonstrate our approach. The smart contract is adapted from the `CreditDAO` contract from Etherscan, where functions are simplified, variables are renamed, and the code is pretty printed for better readability.

usage of some contracts actually requires making significant changes to the underlying *data layout*, a problem that is not addressed by any prior work.

To gain some intuition about optimizations related to data layout, consider the two implementations of the `CreditDAO` contract in Figure 1. Here, the contract on the left uses a struct called `Election` that contains information relevant to an election, such as candidates and their votes. On the other hand, the contract on the right implements the same functionality using different structs. By refactoring the data types in this manner and changing the implementation accordingly, the gas usage of the `CreditDAO` contract can be reduced by approximately 30%. However, in order to obtain such gas savings, developers need to experiment with different layouts, which requires not only changing the data structures but *also* re-implementing significant parts of the contract code.

Motivated by this observation, this paper presents a new technique, and its implementation in a tool called Solidare, for automating such data structure refactoring tasks in smart contracts. Given the source code of a smart contract (implemented in Solidity) and a desired data type refactoring operation, our method automatically generates a functionally equivalent contract that uses the refactored data types. Using our proposed approach, smart contract developers can quickly try out many different data representations and measure the corresponding gas usage. Thus, Solidare allows developers to evaluate many different data representations without having to re-implement their smart contract.

To automate such data type refactoring tasks, Solidare employs three new ideas:

(1) First, Solidare provides a domain-specific language (DSL) for specifying type-level transformations that are useful for reducing gas usage in smart contracts. While such transformations may

require significant changes to the contract's underlying implementation, they can be expressed in just a few lines of code in this DSL.

(2) Second, Solidare automatically generates a *contract sketch* based on the user-provided DSL program. Such a sketch encodes a space of well-typed contract implementations that are consistent with the specified refactoring.

(3) Third, Solidare uses a novel *optimal program synthesis* technique to find a completion of the sketch that is both equivalent to the original contract and that minimizes gas usage (with respect to a proxy cost model) under the specified refactoring. In particular, Solidare tackles this challenging optimal synthesis problem through reduction to *maximum satisfiability (MaxSAT)*.

Since Solidare can *automatically* generate an equivalent and gas-efficient code for a specified refactoring, it allows programmers to efficiently experiment with many different layouts. Moreover, to reduce manual effort even further, Solidare also incorporates an *auto-tuner* that tries to automatically find a good refactoring.

We have used Solidare to automatically refactor data types in real-world smart contracts collected from Etherscan and GasStation and show that our technique is useful for improving gas usage of those contracts that use rich data structures. In particular, using Solidare, we were able to reduce the gas usage of some Solidity programs by up to 48.6%. We also compare our proposed optimal synthesis technique against a baseline that uses enumerative search and show that our proposed ideas are important for scaling this technique to real-world contracts.

In summary, this paper makes the following contributions:

- We propose a technique for optimizing the gas usage of smart contracts based on a convenient domain-specific language for data type refactoring.
- We show how to generate *contract sketches* for a given refactoring and present a novel optimal synthesis algorithm for producing functionally-equivalent programs that minimize a proxy metric for gas usage.
- We experimentally evaluate our technique on smart contracts from Etherscan and GasStation and show that our approach (a) can successfully automate data layout transformations and (b) is useful for reducing gas usage in contracts that have rich data layouts.

2 OVERVIEW

In this section, we illustrate our approach using the motivating example from Figure 1.

2.1 Usage Scenario

Consider a smart contract developer who wants to optimize the gas usage of the CreditDAO contract on the left side of Figure 1. As mentioned in Section 1, the gas usage of this contract can be significantly reduced by changing the data layout and refactoring some of the fields in the original `Election` struct into two new structs called `Count` and `Participant`. However, as is evident by the "diff" between the two CreditDAO versions in Figure 1, this type-level refactoring requires making changes to the code.

Smart contract developers can use our tool, Solidare, to automatically derive the CreditDAO implementation on the right of Figure 1 from its original version on the left. [1] To use Solidare, the developer needs to provide the desired type-level refactoring as a simple program in the Solidare DSL. For instance, in our running example, the desired data representation can be obtained from the original one by (1) moving some of the fields in the `Election` struct to a new struct called `Count`, and (2) introducing a new struct called `Participant` that has two boolean fields. This type-level

[1] Solidare introduces auto-generated names for newly-introduced identifiers (e.g., variable and field names). Developers may choose to rename them if they prefer more human-readable names.

transformation can be specified using the following program in Solidare's DSL :

```
Election, Count = SPLIT(Election, 5);
Participant = WRAP(bool, bool);
```
(1)

Here, the first line tells Solidare to split the `Election` struct into two separate structs called `Election` and `Count` with the first five fields remaining in `Election`. Next, the second line tells Solidare to introduce a new struct called `Participant` with two boolean fields.[2] Overall, this refactoring ends up reducing gas usage for the following reasons:

(1) First, since the fields `numMaxVotes` and `idProcessed` are not accessed as frequently as the other fields, placing them in a separate struct helps avoid unnecessary reads from the blockchain.
(2) Second, by introducing a new `Participant` struct with two booleans, we can merge the two mappings used in the `Election` struct into a single mapping. This transformation ends up being helpful because it reduces the number of read and write operations involving the blockchain.

Interestingly, both of these transformations are necessary for reducing the contract's gas usage. In particular, we observe that each individual transformation on its own does not lead to noticeable gas savings. As illustrated by this example, the impact of a given transformation can be unintuitive in terms of its impact on the contract's gas usage. Thus, it is important to provide smart contract developers with tools that can help them quickly try out different data layouts and observe their impact on gas usage. Solidare serves exactly this purpose: given a transformation expressed in Solidare's DSL, it automatically generates equivalent code using the new data types. Furthermore, using Solidare's auto-tuner, programmers can use our tool to automatically discover new data layouts that reduce gas usage for a given workload.

2.2 How Solidare works

We now explain how Solidare synthesizes a new program given the transformation from Eq. 1 and the original implementation.

Type declarations. First, Solidare generates new data types based on the specified refactoring. For our example, it generates three structs `Election`, `Count`, and `Participant` shown in lines 2–15 on the right side of Figure 1. Beyond introducing new structs, observe that the `candidates` and `userHasVoted` fields inside the original `Election` struct have been merged into a single one in the new contract (line 5 on the right side of Figure 1).

Variable declarations. Next, Solidare modifies and introduces variable declarations as necessary. For this example, it does not change the existing declarations in the contract but adds the following new variable[3]:

```
mapping(uint => Count) counts;
```

In particular, since two of the fields in the original `Election` struct have been moved to a new `Count` struct, we need to introduce a new mapping of type `uint => Count` to preserve all data present in the original `elections` mapping.

Code generation overview. Having transformed the data types and variables, our next step is to generate semantically equivalent code that uses the new data types. However, this problem turns out to be tricky both from a correctness and gas optimality perspective. To gain some intuition about why this is the case, consider the expression `elections[nextEId-1]` that appears in the source program of Figure 1. Since the `Election` struct has been split into two, the `elections`

[2] In general, one may not want to wrap all instances of a type into a struct. To deal with such scenarios, Solidare provides a `typedef` mechanism for creating type aliases, so users can specify which instances of the type they want to transform.
[3] In general, Solidare uses generic names for freshly introduced variables; we refer to it as `counts` here for clarity.

```
function finishElections(uint _iterations) {
    uint currentVotes; Election e; Count c;
    if (??1) e = ??2;
    if (??3) c = ??4;
    uint nextId = ??5;
    for (uint cnt = 0; cnt < _iterations; cnt++) {
        currentVotes = ??6;
        if (currentVotes > ??7) ??8 = currentVotes;
        nextId++;
    }
    ??9 = nextId;
}
```

Fig. 2. Contract sketch.

mapping has now been replaced with two different mappings: `elections` and `counts`. So, how do we know what the right "translation" is for this expression? It could involve just `elections` or just `counts`, or possibly both (if fields from both split parts are accessed later on in the code). We could statically analyze the rest of the code to figure out a likely replacement for this expression, but given that static analysis is imperfect, there will always be some ambiguity. Furthermore, in practice, such correctness and efficiency issues become even more tricky for data type refactorings that introduce new references (e.g., by wrapping primitives into a struct) or eliminate references (e.g., by unwrapping a struct into multiple primitive types).

To deal with these problems, Solidare utilizes *optimal program synthesis* to rewrite the code. In particular, it first generates a sketch whose completions are all well-typed with respect to the specified refactoring. Then, to ensure that the generated code is *both* correct *and* gas efficient, Solidare searches for a sketch completion that is semantically equivalent to the original contract and that minimizes a quantitative objective that serves as a proxy for gas usage.

Sketch generation. Solidare generates a contract sketch by analyzing one method at a time. In particular, Solidare identifies all expressions whose types are affected by the data type refactoring, replaces those expressions with holes, and, for each hole, it computes a *domain* denoting all expressions that can be used to fill that hole. Furthermore, if a statement s contains a hole, we make s optional by introducing a boolean guard: the intuition is that some statements may become redundant as a result of the type refactoring; thus, making these statements optional provides valuable optimization opportunities.

For instance, for the `finishElections` procedure, Solidare produces the sketch shown in Figure 2. Here, since the `Election` struct in the original contract has been split into two parts, we *may* need to introduce both an `Election` variable as well as one of type `Count`; hence, the second and third lines are guarded by a boolean indicating that these assignments may be optional. All other holes (`??5, ??6, ??7, ??8, ??9`) denote expressions that have been potentially affected by the refactoring and that may need to be rewritten.

Sketch completion. Once Solidare generates a sketch, it searches for a completion of the sketch that is functionally equivalent to the original program. A *completion* of the sketch is an assignment from each hole in the sketch to a concrete program expression. Solidare encodes the space of all possible sketch completions as a SAT formula such that every model of this formula corresponds to a candidate solution. Then, Solidare performs synthesis by repeatedly sampling models of this SAT formula, checking for equivalence, and adding blocking clauses to the SAT encoding.

Observe that, in general, there are multiple sketch completions that are functionally equivalent to the original contract, but some of these are more gas-efficient than others. Since our goal is to reduce gas usage, our method augments the SAT encoding with *soft clauses* that serve as a proxy

$$
\begin{array}{rcl@{\qquad}rcl}
\text{Program } \mathcal{P} & ::= & (\Sigma, \Gamma, V, F+) & \text{Function } F & ::= & \mathsf{function}\ f(\vec{e})\ s\ \mathsf{returns}\ \vec{y} \\
\text{StructEnv } \Sigma & ::= & S \rightarrow (\tau_1, \ldots, \tau_r) & \text{Statement } s & ::= & \mathsf{skip} \mid l := e \mid s; s \\
\text{TypeEnv } \Gamma & ::= & x \rightarrow \tau & & \mid & \mathsf{if}(e)\ \mathsf{then}\ s\ \mathsf{else}\ s \mid \mathsf{while}(e)\ \mathsf{do}\ s \\
\text{WordType } W & ::= & \mathsf{int} \mid \mathsf{uint} \mid \mathsf{address} \mid \ldots & \text{LHSExpr } l & ::= & x \mid l.f \mid l[e] \\
\text{Type } \tau & ::= & W \mid S \mid \mathsf{mapping}(W \Rightarrow \tau) & \text{RHSExpr } e & ::= & l \mid c \mid op(\vec{e}) \mid S(\vec{e}) \mid f(\vec{e})
\end{array}
$$

$$x \in \textbf{Variable} \quad c \in \textbf{Constant} \quad S \in \textbf{StructName}$$

Fig. 3. Syntax of smart contracts. All variables are assumed to have globally unique names.

for minimizing gas usage.[4] Thus, Solidare formulates this optimal synthesis task as a MaxSAT problem and ensures that completions of the sketch are explored in order of their gas efficiency according to our cost model. Hence, due to the use of MaxSAT, Solidare can terminate its search as soon as it finds a program that is original to the equivalent one. For instance, for our running example, Solidare produces precisely the desired program from the right side of Figure 1.

Auto-tuning. In this section, we illustrated our approach using a manually-supplied refactoring program. However, as mentioned in Section 1, our implementation also incorporates an auto-tuner that automatically explores candidate refactorings in the DSL and measures their gas usage for a given workload. As we demonstrate empirically in Section 7.5, our auto-tuner can often find refactorings that are competitive with (and, in fact, sometimes better than) manually-written refactorings that we came up with.

3 PRELIMINARIES

In this section, we provide some necessary background on the Solidity language, definition of equivalence, and the Ethereum Virtual Machine.

3.1 Solidity Smart Contracts

In this paper, we consider smart contracts implemented in Solidity, a statically-typed object-oriented language with additional features targeting the Ethereum Virtual Machine (EVM). To simplify presentation, we consider the core subset of Solidity shown in Figure 3. Here, we model a smart contract $\mathcal{P}$ as a tuple $(\Sigma, \Gamma, V, F+)$, where:

- Σ is a so-called *structure environment* mapping each struct name to its corresponding definition, represented as a tuple of types $(\tau_1, \ldots, \tau_n)$;
- Γ is a *type environment* mapping variables (both fields and local variables) to their types;
- $V \subseteq dom(\Gamma)$ are so-called *blockchain variables*;
- $F+$ is a set of functions that can be invoked by users. Function bodies consist of a sequence of statements, including assignments, loads, stores, conditionals, and loops.

In our formalization, we assume variables have globally unique names, and we differentiate between blockchain variables which are stored on the blockchain and local variables that are stored in memory. Since accessing variables stored on the blockchain uses a lot more gas, this distinction gives us a way to quickly estimate a contract's rough gas usage.

Types. Solidity's type system consists of three basic building blocks, namely, *primitive (word) types* such as `uint` and `address`, structs S, and mappings $\mathsf{mapping}(W \Rightarrow \tau)$. We discuss structs and mappings in more detail below.

Structs. A struct S is a named tuple $(\tau_1, \ldots, \tau_n)$ where τ_i denotes the type of the i-th field in S. To simplify presentation, we omit explicit field names and write $v.f_i$ to denote the value of the i-th

[4]Since gas usage is hard to estimate statically, our optimization objective is based on syntactic features (namely, minimizing the number of statements and used blockchain variables) that serve as a simple proxy for gas usage.

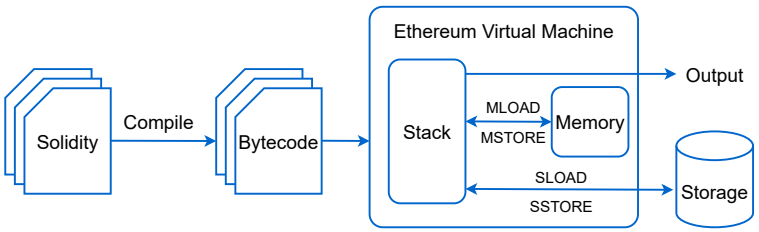

Fig. 4. Ethereum Virtual Machine

field in struct S. As in Solidity, we use the notation $S(e_1, \ldots, e_n)$ to construct a new struct whose i-th field has value e_i. In the rest of this paper, we assume that all structs are references; thus, the expression $S(e_1, \ldots, e_n)$ implicitly performs dynamic memory allocation, and a field access operation $v.f_i$ implicitly dereferences a pointer.

Mappings. In Solidity, a variable of type $\mathsf{mapping}(W \Rightarrow \tau)$ represents a key-value store where the key has (primitive) type W and the value has type τ. Since Solidity mappings are always references, any variable of type $\mathsf{mapping}(W \Rightarrow \tau)$ is dynamically allocated, and any map access $m[e]$ performs a pointer dereference. Note that arrays are a special case of mappings with key type **uint**.

Type expressions. In the rest of this paper, we use the notation ξ to denote a type expression with a single *hole* (denoted $\cdot$), and we write $\xi(\tau)$ to denote the hole in ξ filled with type τ.[5] For instance, if ξ is $\mathsf{mapping}(\mathsf{uint} \Rightarrow \cdot)$, then $\xi(\mathsf{address})$ is $\mathsf{mapping}(\mathsf{uint} \Rightarrow \mathsf{address})$. We also lift this notation to type sequences and write $\xi(\vec{\tau})$ to denote $(\xi(\tau_1), \ldots, \xi(\tau_n))$, where $\vec{\tau} = (\tau_1, \ldots, \tau_n)$.

3.2 Equivalence of Smart Contracts

Since our goal is to refactor a smart contract into an equivalent (and more gas-efficient) version, we first describe what it means for two smart contracts to be equivalent. To this end, we introduce the notion of a *transaction sequence*. Given a contract $\mathcal{P} = (\Sigma, \Gamma, V, \vec{F})$, a *transaction sequence* on $\mathcal{P}$ is a sequence of method invocations of the form:

$$t = (f_1, \theta_1), (f_2, \theta_2), \ldots, (f_n, \theta_n)$$

where f_i denotes the name of a function in $\vec{F}$ and θ_i is the actual parameters for invoking f_i.[6] We also use the notation $[\![\mathcal{P}]\!]_t$ to represent the result of executing transaction sequence t on $\mathcal{P}$.

Given two smart contracts $\mathcal{P}$ and $\mathcal{P}'$ that have the same functions (but possibly with different implementations using different data structures), we consider $\mathcal{P}$ and $\mathcal{P}'$ to be semantically equivalent, denoted $\mathcal{P}' \simeq \mathcal{P}$, if $[\![\mathcal{P}]\!]_t = [\![\mathcal{P}']\!]_t$ holds for all possible transaction sequences t. Intuitively, equivalence in this context means that the contracts produce the same observable results when used in the same manner. We note that this definition of equivalence is the standard notion of *observational equivalence* between two abstract data types [Mitchell 1991].

3.3 Ethereum Virtual Machine

Smart contracts written in Solidity can execute on a variety of different blockchain platforms, of which Ethereum is the most common. In this section, we describe the architecture of the Ethereum Virtual Machine (EVM) to motivate the need for data type refactoring.

Figure 4 gives a schematic overview of the Ethereum Virtual Machine [Buterin 2014; Ethereum 2022; Wood 2022], which is a stack machine that supports general-purpose computation (e.g. ADD,

[5] We disallow the hole from being a key type in mappings.
[6] We assume fall back functions are explicitly listed in the transaction sequence.

$$
\begin{array}{rcl}
\text{Trans. } \mathcal{T} & ::= & s \mid \mathcal{T}; \mathcal{T} \\
\text{Stmt. } s & ::= & S \leftarrow \mathsf{Wrap}(\tau, \ldots, \tau) \mid \mathsf{Unwrap}(S) \mid (S, S) \leftarrow \mathsf{Split}(S, c) \\
& \mid & S \leftarrow \mathsf{Merge}(S, S) \mid S \leftarrow \mathsf{Reorder}(S, c, c) \\
& & c \in \textbf{Constant} \quad S \in \textbf{StructName} \quad \tau \in \textbf{Type}
\end{array}
$$

Fig. 5. Syntax of the transformation language.

NOT) and blockchain-specific operations such as BALANCE, ADDRESS, etc. Executing each instruction on the EVM costs a certain amount of money, which is measured by a metric called *gas*.

The EVM maintains a transient memory that can be accessed using the MLOAD and MSTORE instructions. It can also interact with persistent storage on the blockchain using SLOAD and SSTORE. The gas consumption of loads and stores is significantly higher than other instructions such as arithmetic [Wood 2022]. Therefore, reducing the number of load and store operations is crucial for reducing the gas consumption of smart contracts executing on the EVM or other similar architectures.

To reduce the number of load and store operations of smart contracts, the Solidity compiler tries to pack closely located data of short types (e.g., `Unit32`, `Unit64`) into 256-bit words. However, if small chunks of data are spread across different memory or storage locations, the compiler will fail to pack them automatically. Motivated by this problem, Solidare aims to explore different type-level refactorings to change the data layout of smart contracts, thereby providing more opportunities for the compiler to perform gas optimizations.

4 A DOMAIN-SPECIFIC LANGUAGE FOR DATA TYPE REFACTORING

In this section, we present the syntax and semantics of our domain-specific language for expressing data layout transformations.

4.1 Syntax

Figure 5 presents our data type refactoring DSL, where a program $\mathcal{T}$ consists of a sequence of statements and each statement is one of the following operations:

- The statement $S \leftarrow \mathsf{Wrap}(\tau_1, \ldots, \tau_n)$ creates a new struct S containing fields with types $\tau_1, \ldots, \tau_n$.
- $\mathsf{Unwrap}(S)$ is the inverse of Wrap and removes struct S.
- The statement $(S_1, S_2) \leftarrow \mathsf{Split}(S, n)$ splits the struct S into two separate structs S_1 and S_2 and stores the first n fields of S in S_1 and the remaining fields in S_2.
- Conversely, $S \leftarrow \mathsf{Merge}(S_1, S_2)$ merges all fields in structs S_1 and S_2 into a new struct S.
- $\mathsf{Reorder}(S, i, j)$ swaps the i-th and j-th fields of struct S.

Our DSL is based on the observation that the gas usage of a Solidity program is highly dependent on how data is packed into structs. Thus, all DSL operators modify the layout of data inside structs.

Remark. While Split and Merge are just syntactic sugar for Wrap and Unwrap respectively, we include them in our refactoring DSL for convenience. In particular, for a struct S with m fields of type $\tau_1, \ldots, \tau_m$, the statement $(S_1, S_2) \leftarrow \mathsf{Split}(S, n)$ is shorthand for:

$$\mathsf{Unwrap}(S);\ S_1 \leftarrow \mathsf{Wrap}(\tau_1, \ldots, \tau_n);\ S_2 \leftarrow \mathsf{Wrap}(\tau_{n+1}, \ldots, \tau_m)$$

Similarly, given struct S_1 with n fields of type $\tau_1, \ldots, \tau_n$ and struct S_2 with m fields of type $\tau'_1, \ldots, \tau'_m$, $S \leftarrow \mathsf{Merge}(S_1, S_2)$ is shorthand for:

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

$$\frac{\neg\text{HasFieldSeq}(\mathcal{F}', \xi(\mathcal{F}))}{S \leftarrow \mathsf{Wrap}(\mathcal{F}) \vdash \mathcal{F}' \rightarrowtail \mathcal{F}'} \text{ (Wrap1-Fld)} \qquad \frac{\mathcal{F} = (\tau_1, \ldots, \tau_n) \quad \text{HasFieldSeq}(\mathcal{F}', \xi(\mathcal{F}))}{S \leftarrow \mathsf{Wrap}(\mathcal{F}) \vdash \mathcal{F}' \rightarrowtail \text{Replace}(\mathcal{F}', \xi(\mathcal{F}), \xi(S))} \text{ (Wrap2-Fld)}$$

$$\frac{\neg\text{HasField}(\mathcal{F}, \xi(S))}{\mathsf{Unwrap}(S) \vdash \mathcal{F} \rightarrowtail \mathcal{F}} \text{ (Unwrap1-Fld)} \qquad \frac{\text{HasField}(\mathcal{F}, \xi(S)) \quad \mathcal{F}' = \text{Fields}(S)}{\mathsf{Unwrap}(S) \vdash \mathcal{F} \rightarrowtail \text{Replace}(\mathcal{F}, \xi(S), \xi(\mathcal{F}'))} \text{ (Unwrap2-Fld)}$$

Fig. 6. Auxiliary judgments for field sequences used in Figure 7.

$$\frac{\Sigma = [S_1 \mapsto \mathcal{F}_1, \ldots, S_n \mapsto \mathcal{F}_n] \quad S \leftarrow \mathsf{Wrap}(\mathcal{F}) \vdash \mathcal{F}_i \rightarrowtail \mathcal{F}_i' \quad i \in [1,n]}{S \leftarrow \mathsf{Wrap}(\mathcal{F}) \vdash \Sigma \rightarrowtail [S \mapsto \mathcal{F}, S_1 \mapsto \mathcal{F}_1', \ldots, S_n \mapsto \mathcal{F}_n']} \text{ (Wrap-S)} \qquad \frac{\mathcal{T}_1 \vdash \Sigma \rightarrowtail \Sigma_1 \quad \mathcal{T}_2 \vdash \Sigma_1 \rightarrowtail \Sigma_2}{\mathcal{T}_1; \mathcal{T}_2 \vdash \Sigma \rightarrowtail \Sigma_2} \text{ (Seq-S)}$$

$$\frac{\Sigma = [S_1 \mapsto \mathcal{F}_1, \ldots, S_n \mapsto \mathcal{F}_n] \quad \mathsf{Unwrap}(S_k) \vdash \mathcal{F}_i \rightarrowtail \mathcal{F}_i' \quad i \in [1,n] \setminus \{k\}}{\mathsf{Unwrap}(S_k) \vdash \Sigma \rightarrowtail \Sigma[S_i \mapsto \mathcal{F}_i' \mid i \in [1,n] \setminus \{k\}]} \text{ (Unwrap-S)} \qquad \frac{\mathcal{F} = \Sigma(S) \quad \mathcal{F}' = \text{Swap}(\mathcal{F}, i, j)}{\mathsf{Reorder}(S, i, j) \vdash \Sigma \rightarrowtail \Sigma[S \mapsto \mathcal{F}']} \text{ (RO-S)}$$

Fig. 7. Semantics for structure environments.

4.2 Semantics

The semantics of our DSL is defined over a structure environment Σ and type environment Γ. To facilitate formalization, we first introduce the following auxiliary definitions:

- HasField($\mathcal{F}, \tau$) is true iff field sequence $\mathcal{F}$ contains field τ. Similarly, HasFieldSeq($\mathcal{F}, \mathcal{F}'$) is true iff field sequence $\mathcal{F}$ contains a *consecutive* sub-sequence $\mathcal{F}'$.
- Fields(S) returns the field sequence of struct S.
- Replace($\mathcal{F}, \mathcal{F}_1, \mathcal{F}_2$) yields a field sequence where each occurrence of $\mathcal{F}_1$ in $\mathcal{F}$ is replaced with $\mathcal{F}_2$.
- Swap($\mathcal{F}, i, j$) returns a field sequence with the order of f_i and f_j in $\mathcal{F}$ swapped.

In what follows, we only describe the semantics for Wrap, Unwrap, and Reorder because Split and Merge are just syntactic sugar for Wrap and Unwrap as explained in Section 4.1. Furthermore, we describe the DSL semantics in two parts: first, we explain its effect on structure environment Σ and then its effect on type environment Γ.

DSL semantics on struct definitions. Given struct environment Σ, Figure 7 describes the semantics of a DSL program $\mathcal{T}$ using judgments of the form $\mathcal{T} \vdash \Sigma \rightarrowtail \Sigma'$ where Σ' is the new structure environment obtained by "executing" $\mathcal{T}$ on Σ. These rules utilize the auxiliary judgments in Figure 6 that describe how $\mathcal{T}$ modifies field sequence $\mathcal{F}$.

First, the Wrap1-Fld and Wrap2-Fld rules describe the effects of a statement $S \leftarrow \mathsf{Wrap}(\mathcal{F})$ on a field sequence $\mathcal{F}'$. If $\mathcal{F}'$ contains a sequence of type $\xi(\mathcal{F})$, we replace it with a single field of type $\xi(S)$ (Wrap2-Fld); otherwise $\mathcal{F}'$ remains unchanged as a result of the wrap operation (Wrap1-Fld). Similarly, the two unwrap rules describe the effect of unwrapping struct S on some field sequence $\mathcal{F}$. If $\mathcal{F}$ contains a field of type $\xi(S)$, we replace every occurrence of $\xi(S)$ with $\xi(\mathcal{F}')$ where $\mathcal{F}'$ denotes the fields of S (Unwrap2-Fld); otherwise, $\mathcal{F}$ remains unchanged (Unwrap1-Fld).

Next, the rules in Figure 7 describe the semantics of DSL statements on a structure environment Σ. The first rule (labeled Wrap-S) for $S \leftarrow \mathsf{Wrap}(\mathcal{F})$ first introduces a new struct S with fields $\mathcal{F}$ and then modifies all the struct definitions $S_i \in \Sigma$ by updating their fields according to Figure 6.

Example 4.1. Consider the following struct definition:

```
struct Items { mapping (uint => address) owner;
  mapping (uint => intX) x; mapping (uint => intY) y; }
```

$$\frac{\Gamma' = \Gamma[v \mapsto \xi(S) \mid \exists i.\ \Gamma(v) = \xi(\tau_i)]}{S \leftarrow \mathsf{Wrap}(\tau_1, \ldots, \tau_n) \vdash \Gamma \rightarrowtail \Gamma'}\ \text{(Wrap-T)} \qquad \frac{}{\mathsf{Reorder}(S, i, j) \vdash \Gamma \rightarrowtail \Gamma}\ \text{(Reorder-T)}$$

$$\frac{\mathsf{Fields}(S) = (\tau_1, \ldots, \tau_n) \quad \Gamma' = \Gamma[v \mapsto \bot \mid \Gamma(v) = \xi(S)] \quad \Gamma'' = \Gamma'[v_i \mapsto \xi(\tau_i) \mid \Gamma(v) = \xi(S), i \in [1, n],\ \text{fresh } v_i]}{\mathsf{Unwrap}(S) \vdash \Gamma \rightarrowtail \Gamma''}\ \text{(Un-T)}$$

$$\frac{\mathcal{T}_1 \vdash \Gamma \rightarrowtail \Gamma_1 \quad \mathcal{T}_2 \vdash \Gamma_1 \rightarrowtail \Gamma_2}{\mathcal{T}_1; \mathcal{T}_2 \vdash \Gamma \rightarrowtail \Gamma_2}\ \text{(Seq-T)}$$

Fig. 8. Semantics for type environments.

where intX and intY are both type aliases for uint256. The transformation Point ← Wrap(intX, intY) modifies the struct definitions as follows:

```
struct Point { intX x; intY y; }
struct Items { mapping (uint => address) owner; mapping (uint => Point) p; }
```

In particular, we introduce a new struct definition for Point and then merge the two mappings inside Items into a single field. The latter is because Items contains two consecutive fields of type ξ(intX) and ξ(intY), which get replaced by a single field ξ(Point) according to rule Wrap2-Fld.

Continuing with Figure 7, the Unwrap-S rule for a statement Unwrap(S_k) removes S_k from the structure environment and modifies the definitions of the other structs in Σ by replacing every occurrence of S_k with its corresponding fields using the unwrap rules from Figure 6.

Example 4.2. Consider the struct definitions:

```
struct Point  { intX x; intY y; }
struct Square { Point start; uint32 len; }
```

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

```
struct Square { intX x; intY y; uint32 len; }
```

Finally, the rule RO-S in Figure 7 describes the semantics of Reorder(S, i, j). Here, we simply modify the definition of struct S by swapping fields f_i and f_j. Finally, the last rule labeled Seq-S sequentially composes two DSL statements in the expected way.

Semantics for type environments. Next, we describe how a DSL program $\mathcal{T}$ modifies the type environment. This is described in Figure 8 using judgments of the form $\mathcal{T} \vdash \Gamma \rightarrowtail \Gamma'$ indicating that type environment Γ is transformed to Γ' by $\mathcal{T}$. According to the first rule, Wrap-T, any variable of type $\xi(\tau_i)$ has its type modified to $\xi(S)$, as τ_i has been wrapped into struct S.

Example 4.3. Consider the transformation Point ← Wrap(intX, intY) and type environment Γ:

$$\Gamma(xs) = \mathsf{mapping}(\text{uint} \Rightarrow \text{intX}) \qquad \Gamma(ys) = \mathsf{mapping}(\text{uint} \Rightarrow \text{intY})$$

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

$$\Gamma'(xs) = \mathsf{mapping}(\text{uint} \Rightarrow \text{Point}) \qquad \Gamma'(ys) = \mathsf{mapping}(\text{uint} \Rightarrow \text{Point})$$

Remark. In this example, one of the variables (xs or ys) may actually be redundant (i.e., unused) in the new version of the program. If this is the case, our implementation removes the unused variable in a post-processing phase. However, we define our DSL semantics to keep both variables 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 struct. Thus, the semantics defined here keeps things simple, while the subsequent optimal synthesis algorithm ensures that the transformed program does not contain redundant variables.

Next, consider the second rule (Un-T) in Figure 8 for an $\mathsf{Unwrap}(S)$ operation where S has fields $\tau_1, \ldots, \tau_n$. First, we obtain a new Γ' by removing all variables v of type $\xi(S)$ from Γ. Then, for every variable v of type $\xi(S)$, we introduce n fresh variables $v_1, \ldots, v_n$ of type $\xi(\tau_1), \ldots, \xi(\tau_n)$ respectively. These new variables are blockchain variables iff the original variable v is stored on the blockchain.

Example 4.4. Consider the type environment Γ:

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

where Point has two fields of type intX and intY. Then, Unwrap(Point) yields the following Γ':

$$\Gamma'(p_1) = \text{intX} \quad \Gamma'(m_1) = \mathsf{mapping}(\text{uint} \Rightarrow \text{intX})$$
$$\Gamma'(p_2) = \text{intY} \quad \Gamma'(m_2) = \mathsf{mapping}(\text{uint} \Rightarrow \text{intY})$$

5 CODE SYNTHESIS

In the previous section, we defined the semantics of our type refactoring DSL in terms of its effect on struct definitions and type declarations. In this section, we describe a program synthesis approach to automatically generate a new implementation that uses these types and that is functionally equivalent to the original program. As motivated in Section 2, we take a program synthesis approach to this problem, as it is often unclear how to generate code that is both correct *and* gas-efficient.

5.1 Problem Statement

Given a smart contract $\mathcal{P} = (\Sigma, \Gamma, V, \vec{F})$ and refactoring program $\mathcal{T}$ where $\mathcal{T} \vdash \Sigma, \Gamma \rightarrowtail \Sigma', \Gamma'$, our goal is to automatically generate a new smart contract $\mathcal{P}'$ such that (1) $\mathcal{P}' = (\Sigma', \Gamma', V', \vec{F'})$, (2) $\mathcal{P}$ and $\mathcal{P}'$ are semantically equivalent, i.e., $\mathcal{P}' \simeq \mathcal{P}$, and (3) among all contracts satisfying (1) and (2), $\mathcal{P}'$ should have the lowest gas usage.

Observe that condition (3) in this problem statement requires knowing the gas usage of a smart contract. However, in practice, it is difficult to statically estimate gas usage, and doing so dynamically would be prohibitively expensive. Thus, in this work, we estimate gas usage using a proxy metric Ψ which takes into account both the number of statements and used blockchain variables. In particular, given two contracts $\mathcal{P}_1, \mathcal{P}_2$, we have $\Psi(\mathcal{P}_1) \ \ddot{Y} \ \Psi(\mathcal{P}_2)$ iff:

$$|V_1| \ \ddot{Y} \ |V_2| \vee (|V_1| = |V_2| \wedge \mathit{NumStmts}(\mathcal{P}_1) \ \ddot{Y} \ \mathit{NumStmts}(\mathcal{P}_2)) \tag{2}$$

where V_1, V_2 are the set of *used* blockchain variables used in $\mathcal{P}_1, \mathcal{P}_2$ respectively, and $\mathit{NumStmts}(\mathcal{P})$ yields the total number of statements in $\mathcal{P}$. Intuitively, we wish to minimize the number of blockchain variables used in the contract because operations over blockchain variables are much more expensive than other operations. If the number of blockchain variables is the same, we break ties based on the number of statements. In practice, we found this proxy metric to be effective at comparing the quality of different solutions, and it is easy to evaluate statically.

5.2 Approach Overview

One possible way to approach to the problem from Section 5.1 is to use a syntax-based rewriting technique. While reasonable at first glance, such a syntax-directed translation approach poses two key challenges: First, given a refactoring program, there are multiple possible ways to rewrite the Solidity contract, and it is often difficult to determine which of the rewriting strategies would result in equivalent code. For example, if the refactoring splits a struct A into structs B and C, all source expressions of type A need to be changed to a new expression of type B or C. However, even determining the type of the new expression is not straightforward, because it depends on the context of the expression. Second, syntax-directed translation cannot ensure gas-optimality of the generated code, even with respect to our proxy metric. For instance, to ensure soundness in the

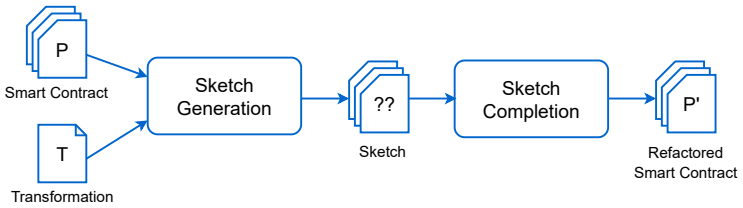

Fig. 9. Workflow of Code Synthesis

presence of aliasing, such a syntax-directed approach would need to perform boxing and unboxing, which may impose extra overhead in terms of gas consumption.

Motivated by these challenges, we instead propose to tackle this problem using a program synthesis approach. Our goal is to solve the following optimal (quantitative) synthesis problem:

*Definition 5.1 (**Synthesis problem**).* Let $\mathcal{P} = (\Sigma, \Gamma, V, \vec{F})$ be a smart contract and $\mathcal{T}$ be a type refactoring program such that $\mathcal{T} \vdash \Sigma, \Gamma \rightarrowtail \Sigma', \Gamma'$. Our synthesis problem is to find a new contract $\mathcal{P}' = (\Sigma', \Gamma', V', \vec{F}')$ such that (a) $\mathcal{P} \simeq \mathcal{P}'$, and (b) $\Psi(\mathcal{P}')$ is minimized, where Ψ is the proxy metric.

Our proposed technique tackles this optimal synthesis problem using a combination of deductive synthesis and search. As shown in Figure 9, we first use deductive (i.e., rule-based) techniques to generate a *program sketch* whose completions are all well-typed with respect to the specified refactoring. In the next phase, we search for a completion of the sketch that is both equivalent to the original contract and that minimizes our proxy metric for gas usage.

5.3 Sketch Generation

First, we discuss how to generate a *contract sketch* given the original program and the new type and variable declarations. A contract sketch is a Solidity program that contains holes, denoted **??**$[\delta]$, indicating unknown expressions that can be instantiated with any of the expressions $e \in \delta$. Here, we refer to δ as the *domain* of its corresponding hole.

At a high level, the basic idea underlying our sketch generation procedure is as follows: First, we identify all expressions that are no longer valid with respect to the new type definitions and type environment — we refer to such expressions as "stale". Then, we replace each stale expression with a set of new expressions under which the program type checks.

Definition 5.2. **(Stale expression)** Let $\mathcal{P} = (\Sigma, \Gamma, V, \vec{F})$ be a smart contract, and let $\mathcal{T}$ be a data type refactoring program such that $\mathcal{T} \vdash \Gamma \rightarrowtail \Gamma'$. We say that an expression e in $\mathcal{P}$ is *stale* with respect to $\mathcal{T}$ if $\Gamma \vdash e : \tau$ but $\Gamma' \not\vdash e : \tau$.

Example 5.3. For the refactoring from Example 4.3, $xs[0]$ is a stale expression because $\Gamma \vdash xs[0] :$ intX but $\Gamma' \vdash xs[0] :$ Point. Similarly, in Example 4.4, the expression p is stale because there is no longer a variable called p.

Type correspondence. Next, we define a correspondence relation B, presented in Figure 10, between types in the original contract and types in the refactored contract. Intuitively, $\tau \mathsf{B} \tau'$ indicates that type τ in the original contract maps to τ' in the refactored contract, and we refer to τ' as the *replacement type* for τ. As shown in Figure 10, replacement types are defined using judgments of the form $\mathcal{T}_A \vdash \tau \mathsf{B} \tau'$ where $\mathcal{T}_A$ is a single statement in the refactoring DSL.

According to the first rule (Wrap-T1), the replacement for any type τ_i that is used as an argument of Wrap is itself. In particular, since type τ_i is still valid, any expression of type τ_i in the original program should be replaced with an expression of the same type. However, while the *type* of the expression remains unchanged, the expression itself does need to be replaced because variables of

$$\frac{i \in [1,n]}{S \leftarrow \mathsf{Wrap}(\tau_1,\ldots,\tau_n) \vdash \tau_i \;\mathsf{B}\; \tau_i}\ (\text{Wrap-T1}) \qquad \frac{i \in [1,n] \quad \xi(x) < x}{S \leftarrow \mathsf{Wrap}(\tau_1,\ldots,\tau_n) \vdash \xi(\tau_i) \;\mathsf{B}\; \xi(S)}\ (\text{Wrap-T2})$$

$$\frac{\text{Fields}(S) = (\tau_1,\ldots,\tau_n) \quad i \in [1,n]}{\mathsf{Unwrap}(S) \vdash \tau_i \;\mathsf{B}\; \tau_i}\ (\text{Unwrap-T1}) \qquad \frac{\text{Fields}(S) = (\tau_1,\ldots,\tau_n)}{\mathsf{Unwrap}(S) \vdash \xi(S) \;\mathsf{B}\; (\xi(\tau_1),\ldots,\xi(\tau_n))}\ (\text{Unwrap-T2})$$

Fig. 10. Definition of replacement type relation.

$$\frac{e \in \{x,\ l.f,\ l[a]\} \quad \text{Stale}(e) \quad \Gamma \vdash e : \tau \quad \mathcal{T}_A \vdash \tau \;\mathsf{B}\; \tau' \quad \delta = \{e' \mid \Gamma' \vdash e' : \tau'\}}{\mathcal{T}_A, \Gamma, \Gamma' \vdash e \quad ??[\delta]}\ (\text{Stale1})$$

$$\frac{e \in \{x,\ l.f,\ l[a]\} \quad \neg\text{Stale}(e)}{\mathcal{T}_A, \Gamma, \Gamma' \vdash e \quad e}\ (\text{NStale}) \qquad \frac{\delta' = \{e' \mid \mathcal{T}_A, \Gamma, \Gamma' \vdash e \quad e',\ e \in \text{AllDoms}(\delta)\}}{\mathcal{T}_A, \Gamma, \Gamma' \vdash ??[\delta] \quad ??[\delta']}\ (\text{Hole})$$

$$\frac{e \in \{x,\ l.f,\ l[a]\} \quad \text{Stale}(e) \quad \Gamma \vdash e : \tau \quad \mathcal{T}_A \vdash \tau \;\mathsf{B}\; (\tau'_1,\ldots,\tau'_n) \quad \delta_i = \{e' \mid \Gamma' \vdash e' : \tau'_i\} \quad i \in [1,n]}{\mathcal{T}_A, \Gamma, \Gamma' \vdash e \quad (??[\delta_1],\ldots,??[\delta_n])}\ (\text{Stale2})$$

$$\frac{\in \{op, S, f\} \quad \mathcal{T}_A, \Gamma, \Gamma' \vdash e_i \quad e'_i \quad i \in [1,n]}{\mathcal{T}_A, \Gamma, \Gamma' \vdash \quad (e_1,\ldots,e_n) \quad (e'_1,\ldots,e'_n)}\ (\text{Comp})$$

Fig. 11. Rules for generating sketch expressions.

this type have been wrapped inside a struct. The second rule (Wrap-T2) deals with mappings that contain a nested type τ_i that has been wrapped. Since our refactoring DSL converts variables of type $\xi(\tau_i)$ to type $\xi(S)$, the replacement type for $\xi(\tau_i)$ is $\xi(S)$.

The next rule (Unwrap-T1) is for types τ_i that appear within a struct that has been unwrapped. While an expression e of type τ_i should retain its old type (i.e., $\tau_i \;\mathsf{B}\; \tau_i$), e still needs to be replaced with a different expression, as it no longer appears inside a parent struct. Finally, Unwrap-T2 handles structs that have been unwrapped. Since there is no longer a struct S in the program, the replacement type for S is a tuple containing fields within S. Similarly, any mapping that contains a value of type S is converted to a tuple of mappings.

Sketch generation for expressions. As mentioned earlier, the key idea behind sketch generation is to replace each stale expression in the program with a hole whose domain includes well-typed expressions. Towards this goal, we define a notion of *valid replacement* for each program expression.

Definition 5.4. **(Valid replacement)** Let $\mathcal{T}_A$ be a type refactoring for $\mathcal{P} = (\Sigma, \Gamma, V, \vec{F})$ such that $\mathcal{T}_A \vdash \Sigma, \Gamma \rightarrowtail \Sigma', \Gamma'$. We say that an expression e' is a *valid replacement* for a stale expression $e \in \mathcal{P}$ if (1) $\mathcal{T}_A \vdash \tau \;\mathsf{B}\; \tau'$ and (2) $\Gamma' \vdash e' : \tau'$.

In other words, a valid replacement for a stale expression e of type τ is another expression e' of type τ' such that τ' is the replacement type for τ. Note that valid replacements are not unique, and there may be multiple valid replacements for a given expression in the source program.

Next, we use this notion of *valid replacement* to describe our sketch generation procedure in Figure 11. The key idea is to replace each stale expression e of type τ with a hole whose domain includes all valid replacements for e (Stale1).[7] The second rule (Stale2) generalizes this idea to a tuple of holes in the case where the replacement type for τ is a tuple. For complex expressions (e.g., involving arithmetic operators or function calls), we recursively generate sketches for nested expressions and then compose them together (rule Comp). Finally, the last rule labeled Hole

[7]In practice, including all valid replacements in the domain is both unnecessary and expensive — we discuss finer-grained domain generation in Section 6.

$$\frac{\neg\text{HasStaleExpr}(l) \quad \neg\text{HasStaleExpr}(e)}{\mathcal{T}_A, \Gamma, \Gamma' \vdash l := e \quad\quad l := e}\ (\text{Assign1}) \qquad \frac{\text{HasStaleExpr}(l) \quad \text{HasStaleExpr}(e) \quad \mathcal{T}_A, \Gamma, \Gamma' \vdash l \quad\quad l' \quad \mathcal{T}_A, \Gamma, \Gamma' \vdash e \quad\quad e'}{\mathcal{T}_A, \Gamma, \Gamma' \vdash l := e \quad\quad l' \ \texttt{J}\ e'}\ (\text{Assign2})$$

$$\frac{\text{HasStaleExpr}(l) \quad \mathcal{T}_A, \Gamma, \Gamma' \vdash l \quad\quad (l'_1, \ldots, l'_n) \quad \text{HasStaleExpr}(e) \quad \mathcal{T}_A, \Gamma, \Gamma' \vdash e \quad\quad (e'_1, \ldots, e'_n)}{\mathcal{T}_A, \Gamma, \Gamma' \vdash l := e \quad\quad l'_1 \ \texttt{J}\ e'_1; \ldots; l'_n \ \texttt{J}\ e'_n}\ (\text{Assign3}) \qquad \frac{\mathcal{T}_A, \Gamma, \Gamma' \vdash s_1 \quad\quad s'_1 \quad \mathcal{T}_A, \Gamma, \Gamma' \vdash s_2 \quad\quad s'_2}{\mathcal{T}_A, \Gamma, \Gamma' \vdash s_1; s_2 \quad\quad s'_1; s'_2}\ (\text{Seq})$$

$$\frac{\mathcal{T}_A, \Gamma, \Gamma' \vdash e \quad\quad e' \quad \mathcal{T}_A, \Gamma, \Gamma' \vdash s_1 \quad\quad s'_1 \quad \mathcal{T}_A, \Gamma, \Gamma' \vdash s_2 \quad\quad s'_2}{\mathcal{T}_A, \Gamma, \Gamma' \vdash \textsf{if}\,(e)\ \textsf{then}\ s_1\ \textsf{else}\ s_2 \quad\quad \textsf{if}\,(e')\ \textsf{then}\ s'_1\ \textsf{else}\ s'_2}\ (\text{Cond}) \qquad \frac{\mathcal{T}_A, \Gamma, \Gamma' \vdash e \quad\quad e' \quad \mathcal{T}_A, \Gamma, \Gamma' \vdash s \quad\quad s'}{\mathcal{T}_A, \Gamma, \Gamma' \vdash \textsf{while}(e)\ \textsf{do}\ s \quad\quad \textsf{while}(e')\ \textsf{do}\ s'}\ (\text{Loop})$$

Fig. 12. Rules for generating sketch statements. The notation l J e is shorthand for the optional assignment **if**(**??**[{⊤, ⊥}]) **then** $l := e$ **else skip**.

constructs sketches for expressions nested inside each hole and then flattens the nested holes to obtain a valid sketch. (This last rule is necessary for handling multi-statement refactorings.)

Example 5.5. Consider the transformation from Example 4.3 and the expression $xs[0] + ys[0]$. According to our sketch generation rules, we replace this expression with $??_1 + ??_2$, where the domain of hole $??_1$ includes $xs[0].f_1$ and $ys[0].f_1$ and the domain of $??_2$ includes $xs[0].f_2$ and $ys[0].f_2$. Note that the domain of $??_1$ does not include $xs[0].f_2$ since the second field of `Point` is `intY`, which is not a valid replacement type for `intX`.

Sketch generation for statements. Next, Figure 12 presents our statement-level sketch generation procedure for atomic transformations. The basic idea is to replace each statement s that contains a stale expression with a new statement $\textsf{if}\,(??[\{\top, \bot\}])\ \textsf{then}\ s'\ \textsf{else skip}$ where s' is obtained from s by replacing its stale expressions with holes. Since the goal of data type refactoring is to improve gas performance, some of the statements in the original program can become redundant after the refactoring. Thus, each statement containing a stale expression is made optional in the sketch by introducing an if statement whose else branch is skip.[8] We now explain Figure 12 in more detail.

The first three rules in Figure 12 show how to generate a sketch for assignments $l := e$. If neither l nor e contain a stale expression, the statement remains unchanged (Assign1). If the statement contains a stale expression and the replacement type for l is not a tuple, we generate a conditional assignment l' J e', which is shorthand for $\textsf{if}\,(??[\{\top, \bot\}])\ \textsf{then}\ l' := e'\ \textsf{else skip}$ (Assign2) where l', e' are obtained using the rules from Figure 11. The final assignment rule (Assign3) is similar except that the replacement type for l and e is a tuple type $(\tau_1, \ldots, \tau_n)$. In this case, we generate n optional assignments, one for each element in the tuple. The remaining rules in Figure 12 recursively apply the sketch generation rules to their sub-statements and compose the results.

Example 5.6. Consider the refactoring from Example 4.4 and the statement `m[0]:= Point(x, y)` with $\Gamma(x)$ = intX and $\Gamma(y)$ = intY. We generate the following sketch for this program:

```
if (??1) /* ??1∈{true, false} */  ??2 := ??3; // ??2∈{m1[0],...}, ??3∈{x,...}
if (??4) /* ??4∈{true, false} */  ??5 := ??6; // ??5∈{m2[0],...}, ??6∈{y,...}
```

[8] Our actual implementation may not make certain statements optional if we know they are definitely necessary or the statements are already guarded by an if statement. For example, if there is a backward dependency of the left-hand side of an assignment, we will not make the assignment optional.

$$\frac{\mathcal{T}_A \vdash \Gamma \rightarrowtail \Gamma' \quad \mathcal{T}_A, \Gamma, \Gamma' \vdash s \quad s'}{\mathcal{T}_A, \Gamma, \Gamma' \vdash s \quad {}^* s'} \text{ (Base)} \qquad \frac{\mathcal{T}_A, \Gamma, \Gamma_1 \vdash s \quad {}^* s_1 \quad \mathcal{T}, \Gamma_1, \Gamma_2 \vdash s_1 \quad {}^* s_2}{\mathcal{T}_A; \mathcal{T}, \Gamma, \Gamma_2 \vdash s \quad {}^* s_2} \text{ (Ind)}$$

Fig. 13. Rules for composing refactoring operators.

Algorithm 1 Sketch Completion

```
1: procedure CompleteSketch(𝒮, 𝒫)
   Input: Sketch 𝒮, Source program 𝒫
   Output: Target program 𝒫′ or ⊥ to indicate failure
2:     Φ ← Encode(𝒮);
3:     while SAT(Φ) do
4:         ℳ ← GetModel(Φ);
5:         𝒫′ ← Instantiate(𝒮, ℳ);
6:         if 𝒫′ ≃ 𝒫 then return 𝒫′;
7:         Φ ← Φ ∧ Block(𝒫, 𝒮, ℳ);
8:     return ⊥;
```

Multi-statement refactorings. Figure 13 summarizes how to generalize sketch generation to multi-statement refactorings. In particular, the idea is to first generate a sketch for the first atomic transformation; then apply the next atomic transformation to the generated sketch, and so on.

Theorem 5.7. [9] *Let $\mathcal{P} = (\Sigma, \Gamma, V, \vec{F})$ be a well-typed program and $\mathcal{T}$ be a data type refactoring such that $\mathcal{T} \vdash \Sigma, \Gamma \rightarrowtail \Sigma', \Gamma'$. Given a statement s in $\mathcal{P}$, let $\mathcal{S}$ be the sketch obtained from s (i.e., $\mathcal{T}, \Gamma, \Gamma' \vdash s \quad {}^* \mathcal{S}$). Then, for any mapping σ from holes in $\mathcal{S}$ to expressions in its domain, $\mathcal{S}[\sigma]$ is well-typed under Γ'.*

5.4 Sketch Completion

In this section, we describe our optimal sketch completion algorithm. Given a contract sketch $\mathcal{S}$, the goal is to instantiate each hole $h \in \mathcal{S}$ with an expression $e \in \delta(h)$ such that (1) the resulting program is equivalent to the original contract, and (2) our optimization objective is minimized.

The high-level idea behind this algorithm is to reduce the problem to *maximum satisfiability (MaxSAT)* [Li and Manya 2009], where hard constraints describe the search space and soft constraints enforce the optimality requirement. Given such a MaxSAT encoding, our algorithm repeatedly samples solutions to the MaxSAT formula until it finds a completion that is equivalent to the original contract. Because soft constraints used in the MaxSAT encoding enforce optimality, the algorithm can safely terminate as soon as it finds an equivalent program.

Our top-level sketch completion procedure is presented in Algorithm 1, which first invokes the Encode procedure (line 2) to generate a MaxSAT encoding Φ. This encoding is constructed in such a way that (a) every valid solution of the synthesis problem corresponds to a boolean assignment that satisfies the hard constraints, and (b) if a sketch completion P has lower gas usage than another completion P' with respect to our cost model, then the weight of the satisfied soft constraints for P is higher than those for P'. Once the algorithm constructs this MaxSAT encoding, it enters a loop that repeatedly samples models of Φ (line 4). If the currently sampled model corresponds to a smart contract that is equivalent to the original one, the algorithm returns it as a solution (line 6).

[9] Proofs of all theorems are provided in the extended version of this paper.

Otherwise, the algorithm invokes the Block procedure to prevent the same model (as well as other infeasible ones) in future iterations.

In the remainder of this section, we explain the Encode and Block procedures in more detail. Because checking equivalence between two smart contracts is orthogonal to the main contributions of this paper, we defer a discussion of our equivalence checker to Section 6.

5.4.1 MaxSAT Encoding. The idea behind our MaxSAT encoding is as follows: First, for each hole h_i in the sketch and expression $e_j \in \delta(h_i)$, we introduce a boolean variable b_i^j such that b_i^j is true iff hole h_i is instantiated with expression e_j. Then, since every hole must be instantiated with *exactly one* expression in its domain, we introduce the following hard constraint:

$$\bigwedge_{h_i \in \mathcal{S}} \oplus(b_i^1, \ldots, b_i^{n_i}) \tag{3}$$

where $\oplus$ is an n_i-ary exclusive-or operator and $|\delta(h_i)| = n_i$. In addition, we ensure that different occurrences of source expression e_s are transformed into the same target expression as follows:

$$\bigwedge_{e_j \in \text{doms}(e_s)} \bigwedge_{h_i \in \text{holes}(e_s)} b_i^j \tag{4}$$

Here, $\text{holes}(e_s)$ represents all holes generated from expression e_s and $\text{doms}(e_s) = \cup_{h_i \in \text{holes}(e_s)} \delta(h_i)$. Intuitively, for all holes H generated from the source expression e_s, if a hole $h_i \in H$ is instantiated with candidate expression e_j, then each hole $h_k \in H$ must also be instantiated with e_j.

Example 5.8. Consider the transformation from Example 4.3 and the statements $z = xs[0]; xs[1] = xs[1] + z$ with $\Gamma(z) = \text{intX}$. According to our sketch generation rules, the corresponding sketch is

```
if (??1) /* ??1 ∈ {true, false} */ ??2 := ??3; // ??2, ??3 ∈ {z.f1, xs[0].f1}
if (??4) /* ??4 ∈ {true, false} */ ??5 := ??6+??7; // ??5,??6,??7 ∈ {z.f1, xs[0].f1, xs[1].f1}
```

Here, $??_5$ and $??_6$ are both associated with the source expression $xs[1]$; thus, due to Eq. 4, we add the following hard constraint to our MaxSAT encoding:

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

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

Next, recall from Section 5.1 that our goal is to find a program $\mathcal{P}' = (\Sigma', \Gamma', V', \vec{F}')$ that minimizes the optimization objective from Eq. 2, which consists of a linear combination of the number of used blockchain variables and number of statements. We specify this optimization objective as soft constraints in the MaxSAT encoding.

Minimizing blockchain variables. We first discuss how to encode the first part of the optimization objective (i.e., number of blockchain variables). Let $\text{guards}(h_i)$ denote the set of all boolean holes that h_i is syntactically nested in. If all guards of a hole h_i are assigned to true, then we prefer solutions that do *not* instantiate h_i with an expression containing a blockchain variable. Thus, for *every* blockchain variable v that appears in the domain of some hole, we add the following soft clause with weight n, where n is the number of all possible statements in the contract sketch:

$$\bigwedge_{h_i \in \mathcal{S}} \bigwedge_{h_k \in \text{guards}(h_i)} b_k^1 \rightarrow \bigwedge_{e_j \in \delta(h_i)} \mathsf{I}_v(e_j) \rightarrow \neg b_i^j \tag{5}$$

Here, $\mathsf{I}_v(e)$ is an indicator function stating that variable v is used within expression e. Thus, the soft clause for blockchain variable v states that, for every hole h_i, if the guards of h_i are all assigned to true and expression e_j in the domain of h_i uses blockchain variable v, then the corresponding boolean variable b_i^j should (preferably) be false. Intuitively, for each blockchain variable v, Equation (5) is a

soft constraint encoding that the synthesized program does not use v. If a smart contract has m blockchain variables, there are m soft constraints in total. Since the MaxSAT problem aims to find a solution that can satisfy all hard constraints and maximum weight of soft constraints, the solver will find a solution where the least number of blockchain variables are used by the synthesized program, which is consistent with the objective of the optimization.

Example 5.9. Consider the sketch in Example 5.8 and assume z is a blockchain variable. In the sketch, z occurs in holes $??_2$ and $??_3$ (guarded by $??_1$), as well as holes $??_5, ??_6, ??_7$ that are guarded by $??_4$. Based on Eq. 5, we add the following soft constraint:

$$(b_1^1 \rightarrow (\neg b_2^1 \wedge \neg b_3^1)) \wedge (b_4^1 \rightarrow (\neg b_5^1 \wedge \neg b_6^1 \wedge \neg b_7^1))$$

Assuming that $z.f_1$ is the first element in the domain of holes $??_2, ??_3, ??_5, ??_6, ??_7$, this constraint says the following: If the guards of these holes are enabled (e.g., b_1^1), then they should not be instantiated with $z.f_1$.

Minimizing statements. Next, to encode the second part of the optimization objective (i.e., number of statements). we want to maximize the number of statement guards that are assigned to false. Thus, for each hole h_i representing a statement guard in the sketch, we introduce the following soft clause with weight 1:

$$\hat{\bigcup}_{h_k \in \text{guards}(h_i)} b_k^1 \rightarrow b_i^2 \tag{6}$$

Theorem 5.10. *Given the MaxSAT encoding Φ of a sketch $\mathcal{S}$, the optimal solution to Φ minimizes our proxy gas usage metric Ψ.*

5.4.2 Blocking Clause Generation. We now discuss how to add blocking clauses to the MaxSAT encoding when a candidate sketch completion is incorrect. Given a model $\mathcal{M}$ of Φ, we can avoid getting the same program by just conjoining $\neg\mathcal{M}$ with Φ. While this strategy would work in principle, it is not very effective in practice due to the large search space. Thus, to make synthesis more tractable, we use the notion of *conflict driven learning* that is used heavily in theorem provers [Marques-Silva et al. 2009] and that has recently been applied to program synthesis [Feng et al. 2018; Wang et al. 2019a, 2020]. The key idea is to generalize $\mathcal{M}$ and add a blocking clause that prevents *many* incorrect programs at the same time.

First, somewhat similar to the notion of *minimal unsatisfiable core* [Lynce and Marques-Silva 2004], we introduce a notion called a *minimal failing sub-contract*. To define this notion, let $\mathcal{P} \downarrow F$ denote the same contract as $\mathcal{P}$ but only containing functions F.

Definition 5.11. (**Minimal failing sub-contract**) Given the original contract $\mathcal{P} = (\Sigma, \Gamma, V, F)$ and the candidate refactored contract $\mathcal{P}' = (\Sigma', \Gamma', V', F')$, a *minimal failing sub-contract* is $\mathcal{P}^* = (\Sigma', \Gamma', V', F^*)$ such that (1) $F^* \subseteq F'$; (2) $\mathcal{P} \downarrow F^* \not\simeq \mathcal{P}^*$; (3) for any $\hat{F} \subset F^*$, we have $\mathcal{P} \downarrow \hat{F} \simeq \mathcal{P}' \downarrow \hat{F}$.

In other words, given an incorrect contract $\mathcal{P}'$, a minimal failing sub-contract $\mathcal{P}^*$ of $\mathcal{P}'$ is one that (1) only contains a subset of the functions in $\mathcal{P}'$, (2) the behavior of $\mathcal{P}^*$ is not equivalent to the original contract with respect to the functions it implements, and (3) $\mathcal{P}^*$ 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 5.12. Let us consider a simple 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; } }
```

Algorithm 2 Blocking clause generation

1: **procedure** BLOCK($\mathcal{P}, \mathcal{S}, \mathcal{M}$)
   Input: Source program $\mathcal{P}$, Sketch $\mathcal{S}$, Model $\mathcal{M}$
   Output: Blocking clause φ
2: $(\mathcal{S}', \mathcal{M}') \leftarrow$ MinFailingSubcontract($\mathcal{P}, \mathcal{S}, \mathcal{M}$);
3: $\mathcal{M}^* \leftarrow \{\}$;
4: **for each** $b_i^j \in dom(\mathcal{M}')$ **do**
5: **if** $\forall h_k \in$ guards(h_i). $\mathcal{M}'(h_k^1) = \top$ **then**
6: $\mathcal{M}^* \leftarrow \mathcal{M}^* \uplus [b_i^j \mapsto \mathcal{M}'(b_i^j)]$;
7: $\varphi \leftarrow \hat{\bigvee}_{b \in dom(\mathcal{M}^*)} b < \mathcal{M}^*(b)$;
8: **return** φ;

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 sub-contract of P for three reasons: (1) P^* only contains a subset of functions in P: `set` and `getX`; (2) the behavior of P^* is different from that of P because executing `set(1, 2); getX()` returns 1 on P but returns 2 on P^*; (3) P^* is minimal because removing either function `set` or `getX` results in a sub-contract that is equivalent to the corresponding part of P. For instance, if function *set* is removed from P^*, then `getX` always returns 0 on both P and P^*.

Thus, to generate a good blocking clause, we first compute a minimal failing sub-contract $\mathcal{P}^*$ of the candidate refactoring $\mathcal{P}'$. Since $\mathcal{P}^*$ is already a failing sub-contract, the assignments to the holes in $\mathcal{P}^*$ must already be wrong. Thus, any refactoring that agrees with $\mathcal{P}^*$ on the hole assignments used in $\mathcal{P}^*$ must already be wrong, so, rather than blocking a single implementation $\mathcal{P}'$, we can prune all refactorings that subsume $\mathcal{P}^*$.

Example 5.13. Let us continue with Example 5.12. Now consider a sketch S that is generated from the original program P:

```
contract SimplePoint {
  uint public x = 0; uint public y = 0;
  function set(uint _x, uint _y) public { x = ??1; y = ??2; } // ??1,??2∈{_x, _y}
  function getX() public returns (uint) { return ??3; } // ??3 ∈ {x, y}
  function getY() public returns (uint) { return ??4; } // ??4 ∈ {x, y} }
```

Since `getY` is not part of the minimal failing sub-contract from Example 5.12, the assignment to $??_4$ is irrelevant, meaning that we can prune all sketch completions where $??_1 = \_x, ??_2 = \_y, ??_3 = y$.

Algorithm 2 shows our blocking clause generation technique based on a refinement of this idea. In particular, let $\mathcal{P}' = \mathcal{S}'[\mathcal{M}']$ be a minimal failing sub-contract of $\mathcal{S}[\mathcal{M}]$ as shown in line 2 of Algorithm 2.[10] Now, let us call a hole h in a sketch $\hat{S}$ *enabled* by model $\hat{\mathcal{M}}$ if $\hat{\mathcal{M}}$ assigns all of h's boolean guards to true and disabled otherwise. Then, any model $\mathcal{M}''$ that agrees with $\mathcal{M}'$ on just the assignments to the enabled holes is also guaranteed to yield an incorrect completion. Thus, when generating a blocking clause, we can disregard assignments to all variables that are either (1) not in the domain $\mathcal{M}'$ or (2) disabled by $\mathcal{M}'$.

[10] We describe our technique for computing minimal failing sub-contracts in Section 6.

Theorem 5.14 (Soundness). *Suppose we have a sound and complete oracle for proving equivalence of two programs. Given a source program $\mathcal{P}$ and a sketch $\mathcal{S}$, if CompleteSketch($\mathcal{S}$, $\mathcal{P}$) yields a program $\mathcal{P}'$ where $\mathcal{P}' < \bot$, then $\mathcal{P}' \simeq \mathcal{P}$ and the proxy metric $\Psi(\mathcal{P}')$ is minimized.*

Theorem 5.15 (Completeness). *Suppose we have a sound and complete oracle for proving equivalence of two programs. Given a source program $\mathcal{P}$ and a sketch $\mathcal{S}$, if CompleteSketch($\mathcal{S}$, $\mathcal{P}$) yields $\bot$, then there does not exist completion $\mathcal{P}'$ of $\mathcal{S}$ such that $\mathcal{P}' \simeq \mathcal{P}$.*

6 IMPLEMENTATION AND OPTIMIZATIONS

We have implemented our proposed approach in a tool called Solidare, which is implemented in a combination of Java and Kotlin. Solidare uses the Sat4J [Le Berre and Parrain 2010] tool to solve the generated MaxSAT instances.

Auto-tuner. To reduce developer effort even further, our implementation incorporates an auto-tuner that can *automatically* search for useful data refactorings. In addition to the original contract, the auto-tuner also takes as input a workload $\mathcal{W}$ describing the usage frequency of each function in $\mathcal{P}$ and finds a refactoring program in the DSL from Figure 5 that achieves the best gas usage. To do so, our auto-tuner implementation enumerates DSL programs up to a certain bound, using various heuristics to prioritize refactorings that are more likely to save gas. For instance, we notice that certain constructs in our DSL often need to be used together, e.g., Wrap is typically combined with Reorder, because fields need to be re-organized for alignment and wrapping. Thus, our auto-tuner first performs enumeration based on a library of templates (e.g., Reorder(*); Wrap(*); Reorder(*)) and then falls back on brute-force search. Then, for each enumerated refactoring, the auto-tuner uses our proposed synthesis algorithm to generate a new contract, and, in the end, outputs the program with the best gas usage. Thus, in combination with this auto-tuner, our proposed techniques can allow for a fully automated solution to gas optimization.

Usage modes. Solidare can be used in two modes, namely with and without the auto-tuner. If the auto-tuner is used, programmers need not provide any insights beyond the source program and a representative workload. If the auto-tuner is not used, Solidare requires a data layout transformation for refactoring in addition to the source program. To specify a data layout refactoring that improves gas usage, programmers need some intuition about which layouts are likely to be more gas efficient. But, even in this case, Solidare automates the tedious aspect of rewriting the code and allows programmers to quickly test multiple hypotheses about which data layouts could lead to gas savings.

Type aliases and `typedef`. Solidare provides a `typedef` mechanism for creating type aliases. Since users may not want to apply a transformation to *all* occurrences of a type, they can simply create a type alias and change the particular occurrences they want to refactor to this new type. Hence, this mechanism is important for finer-grained refactorings and provides a sweet spot between type and expression-level refactorings. For example, the user can create an alias called `UintX` for type `unit256`, which allows users to specify particular instances of a type that they want to refactor. Solidare resolves `typedefs` in the pre-processing stage and transform all user-defined type aliases to built-in types of Solidity; the obtained smart contract can be type checked by the Solidity compiler.

Sketch generation. Our implementation performs several optimizations over the sketch generation procedure described in Section 5.3. First, recall that our sketch generation technique uses the notion of *valid replacement expressions* for hole domains, which includes all expressions that are well-typed with respect to the specified refactoring. However, in practice the set of all replacement expressions for a given hole can be very large (or even infinite). In our implementation, the hole domains only

include those expressions that (a) involve a variable v that is affected by the refactoring, and (b) if v corresponds to a mapping or array, then indices must involve expressions that occur in the source. Second, our implementation uses domain knowledge and lightweight static analysis to prune expressions from the domain of each hole. For example, we remove all r-value expressions (e.g. predicates, function calls) from hole domains corresponding to l-value expressions. In addition, recall that our sketch generation makes each statement optional through a boolean guard, but, in some cases, these guards may be redundant. For example, sketch generation might end up introducing nested guards or make optional a statement defining a variable that is used in subsequent code. Our implementation post-processes the generated sketch to remove such unnecessary ambiguity.

Sketch completion. Our implementation optimizes sketch completion by using additional hard constraints in the MaxSAT encoding. For example, we disallow sketch completions where the left- and right-hand-sides of an assignment are identical. As another example, we also disallow sketch completions where a variable is read from without being initialized. Such non-sensical sketch completions are ruled out by adding suitable hard constraints to the MaxSAT instance.

Equivalence checking. Recall that Algorithm 1 needs to check whether a candidate contract is equivalent to the original one. Our implementation leverages the Solis equivalence checker for Solidity [Mariano et al. 2020] but incorporates some optimizations to make it more practical. Specifically, we leverage the observation that many of the candidates proposed by the synthesizer can be refuted using testing. Thus, we first generate a few representative test cases and only invoke the equivalence checker if we cannot refute the candidate program using these test cases.

Generating minimal failing subcontracts. Recall that Solidare uses minimal failing sub-contracts to generate useful blocking clauses. One obvious way to compute these minimal failing sub-contracts is to remove functions from the source and target contracts until the resulting contract satisfies Definition 5.11. However, since this approach requires invoking the equivalence checker during minimization, it can be quite slow. Instead, Solidare leverages the counterexamples provided by the equivalence checker to generate such a failing sub-contract. In particular, the counterexamples returned by the equivalence checker consist of a sequence of function invocations. Since these counterexamples typically involve only a small number of functions, we compute a failing sub-contract by removing the functions that are not used in the counterexample. While this approach definitely produces failing sub-contracts, they may not be *minimal* in principle. Nevertheless, we found this strategy to be very effective in practice.

7 EVALUATION

In this section, we describe a series of experiments that are designed to answer the following research questions:

RQ1. Is Solidare able to generate equivalent code for different data layouts?
RQ2. Can we reduce the gas usage of real-world smart contracts through data type refactoring?
RQ3. How much manual effort does Solidare save developers?
RQ4. How does our sketch completion method compare with simpler baselines?
RQ5. Is Solidare's auto-tuner able to automatically find gas-saving refactorings?
RQ6. How does Solidare compare with other gas optimization tools?

Benchmarks. To answer the research questions above, we collected two sets of benchmarks [11]:

[11] The number of contracts used in our evaluation is in line with recent work [Li et al. 2020]. Furthermore, although there is no fundamental limitation in evaluating Solidare on more benchmarks, constructing realistic workloads and measuring gas usage is a non-trivial and time-consuming endeavor that is hard to scale to a very large number of contracts.

- **Etherscan**: 20 smart contracts from Etherscan [Etherscan 2022] that use rich data structures (e.g., many structs and mappings) and complicated control flows (e.g., loops with large body). Smart contracts in this dataset cover a wide variety of common application scenarios, such as auctions, crowd sourcing, and decentralized autonomous organizations (DAOs).
- **GasStation**: 10 smart contracts from ETH Gas Station [ETHGasStation 2022], which contains a collection of the most frequently used smart conracts. This dataset covers various tokens and decentralized financial applications.

These benchmarks are selected based on the following criteria:

(1) The contract contains at least one loop whose body updates storage (blockchain) variables. Since writing to storage variables is expensive, such contracts have the potential to be optimized.
(2) It contains at least one public function whose instructions are dominated (>50%) by storage accesses. The intuition for this criterion is the same as the previous one.
(3) It contains large structs where at least a field is not word-aligned. Since almost all gas optimizations related to data layout involve structs, this criterion allows us to focus on contracts that may benefit from changing data layout.

Transformations. To perform our evaluation, we manually inspected each program in the Etherscan dataset and came up with a data refactoring that we *thought* might be useful for reducing gas usage. We then used Solidare to generate the new contract and measured gas usage. If gas usage did not improve and we could think of another transformation that might help, we iterated this process for up to three transformations. Thus, each Solidity program comes with 1-3 DSL transformations, giving us a total of 39 benchmarks. For each program in the GasStation dataset, we directly use the best transformation found by the auto-tuner of Solidare. Thus, each Solidity program comes with one transformation, resulting in 10 benchmarks in total.

Experimental Setup. All experiments are conducted on a machine with Intel(R) Xeon(R) E5-2640 @ 2.60GHz CPU and 128 GB of physical memory, running a Docker container with the Ubuntu 18.04 operating system, on Docker version 1.6.2.

7.1 Main Results

Our main experimental results are summarized in Table 1. Here, the columns "Contract", "LOC", and "# Funcs" give statistics about each benchmark (contract name, lines of code, and number of functions). The next column shows the number of DSL transformations we tried for the corresponding Solidity program. The next two columns provide information about the running time of Solidare. Specifically, the columns labeled "Sketch Time" and "Completion Time" show the sketch generation and sketch completion time in seconds averaged across all transformations. Finally, the last two columns provide information about the size of the diff between the original contract and its transformed version. The column labeled "Max Diff" shows the maximum diff (in terms of lines of code) across all transformations, and the one labeled "Avg Diff" shows the average.

The key result is that Solidare is able to generate a semantically equivalent contract for all 49 benchmarks, and its average (resp. median) running time (including both sketch generation and completion) is 21.1 (resp. 0.9) seconds. In most cases, Solidare's running time is dominated by sketch completion, which also includes calls to the equivalence checker.

Outlier analysis. While Solidare is able to generate the new contract quite efficiently in most cases, there is one contract, namely EMPresale, where Solidare takes around 9 minutes to generate the new contract. This is because the size of the search space for the generated sketch is very large ($\approx 8.8 \times 10^{12}$) and the synthesizer ends up exploring 7756 sketch completions before it finds the correct one. There are also a few other outliers (e.g., JanKenPon) where sketch generation takes

Table 1. Statistics about benchmarks and Solidare's running time.

	ID	Contract	LOC	# Funcs	# Trans	Sketch Time (s)	Completion Time (s)	Max Diff	Avg Diff
Etherscan	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
GasStation	21	MetaMasks	597	88	1	0.1	0.2	18	18.0
	22	MoonStaking	525	65	1	0.1	0.4	19	19.0
	23	MoonStakingForTax	842	120	1	0.1	0.3	24	24.0
	24	MASTERPLAN	494	57	1	0.1	0.4	21	21.0
	25	MasterInu	758	149	1	0.1	0.9	15	15.0
	26	MetaPunkController2022	1586	446	1	0.2	0.2	14	14.0
	27	KaijuFrenz	924	99	1	0.1	0.2	25	25.0
	28	EMOBUDDIES	852	101	1	0.1	0.2	15	15.0
	29	GemSwap	528	76	1	0.1	0.3	16	16.0
	30	LL420Reveal	131	16	1	0.1	0.1	10	10.0

longer than sketch completion. Such cases involve very complex expressions in the source contract, so the domain of the holes is quite large. On the other hand, the sketch completion engine is able to find the right instantiation of the holes after only a few iterations.

RQ1: Solidare is able to synthesize an equivalent version of the contract for all benchmarks with an average (resp. median) running time of 21.1 (resp. 0.9) seconds.

7.2 Reduction in Gas Usage

Since the main motivation behind this work is to reduce gas usage, we investigate whether it is possible to obtain significant gas savings by changing the data layout. Similar to prior work [Albert et al. 2020b], we report the savings of execution cost rather than the deployment cost (i.e., cost for uploading code and data to blockchain), as the deployment cost is one-time only.

Measuring gas savings. Since smart contracts are open programs where public functions can be invoked by other smart contracts or account holders, we first construct representative workloads for evaluating execution cost. Specifically, we develop a crawler to extract the transaction history of a smart contract on Etherscan and determine how frequently and in what order the contract's public functions are invoked. Then, based on these measurements, we construct a realistic workload

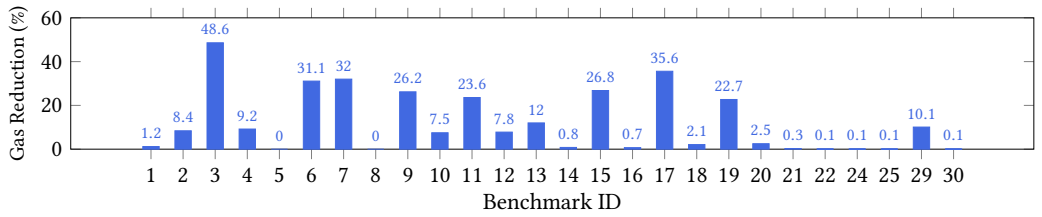

Fig. 14. Gas reduction in benchmarks.

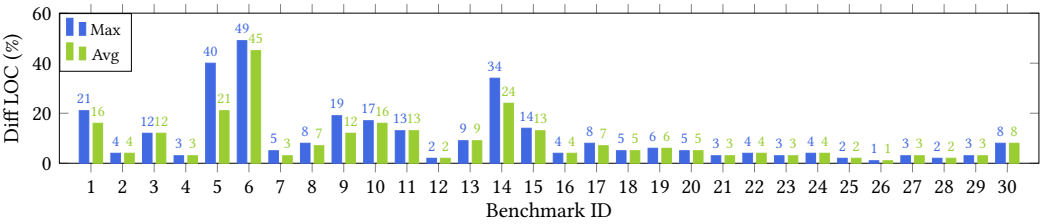

Fig. 15. Diff size as percentage of the lines of code in original contracts.

(a sequence of method invocations) where the frequency and order of methods are the same as those found in the transaction history. To evaluate the gas savings, we measure the gas usage of the original and transformed contract on these workloads.

Results. The results of this evaluation are shown in Figure 14, where the x-axis corresponds to contract IDs and the y-axis shows the reduction in gas usage for the best transformation for that contract. A key take-away is that there is an improvement in gas usage for 18 contracts in the Etherscan dataset, with an average gas savings of 16%. For the GasStation dataset, Solidare can improve gas usage for 6 out of 10 smart contracts, and the gas savings range from 0.1% to 10.1%.

Qualitative analysis. To better understand why Solidare results in better gas savings for the Etherscan dataset, we manually inspect and compare these benchmarks. Most of the smart contracts in GasStation are digital tokens and their supporting libraries. In contrast to other contract types, such as auctions, that require more complex program logic, digital tokens are more straightforward ledger services, so, their data layout is significantly simpler than the contracts in the Etherscan dataset. In particular, the GasStation contracts contain few structs and lots of 256-bit primitive types, such as `unit` and `uint256`. Because such types are already word-aligned, there is little room for optimizing these contracts by changing the data layout.

> **RQ2:** We used Solidare to reduce the gas usage of 18 out of 20 contracts in the Etherscan dataset, with an average reduction of 16%. The gas savings in 9 of the 10 GasStation contracts are not very significant since most types are already word-aligned; however, there is one contract with 10% improvement.

7.3 Savings in Manual Effort

As stated in RQ3, we also wish to understand the manual effort that Solidare saves developers in optimizing gas usage. The ideal way to answer this question is through a user study; however, since we did not have access to smart contract developers, we instead investigate this question through a proxy metric, namely the diff size between the original contract and its refactored version. As mentioned earlier, these statistics are provided in the last two columns of Table 1.

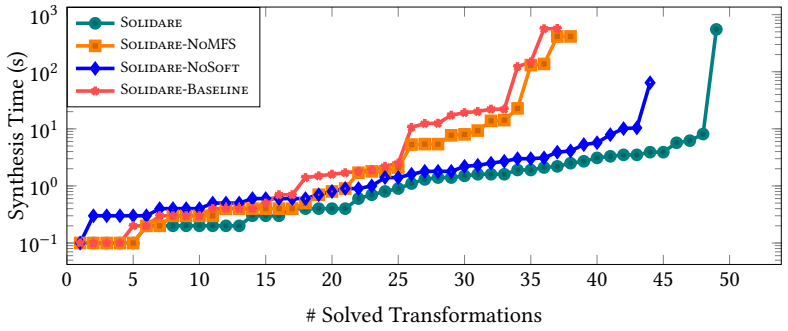

Fig. 16. Comparing Solidare against baselines. y-axis is on log-scale.

In addition, Figure 15 presents the diff between the original and refactored contract in terms of percentage of lines that need to be modified for each Solidity program. Specifically, the blue bars labeled "Max" provide statistics for the transformation that requires the most changes, and the green bars labeled "Avg" show the average diff ratio per benchmark across all transformations. As we can see from Figure 15, some benchmarks require changing a substantial part of the application; in fact, in one case, the diff size is 49% of the original code. Overall, the average number of lines that need to be modified per transformation is 25. However, since it is often unclear what data layout would result in gas savings, one needs to try multiple data layout options, which requires changing the contract in different ways. For instance, across the 20 contracts in the Etherscan dataset, one needs to change a total of 53 lines (for all transformations) *on average*.

RQ3: Solidare can automatically rewrite 10% of lines of code on average for the smart contracts in our benchmark set.

7.4 Ablation Study for Sketch Completion

In this section, we evaluate the effectiveness of our proposed MaxSAT-based sketch completion procedure by performing an ablation study. Specifically, to evaluate RQ4, we compare Solidare against three variants that use a simpler sketch completion procedure:

- Solidare-NoMFS is a variant of Solidare that does not utilize minimal failing sub-contracts to generate blocking clauses. Instead, every time Solidare-NoMFS finds a model M corresponding to a program that cannot be verified, it adds $\neg M$ to the MaxSAT encoding.
- Solidare-NoSoft is a variant that does not use soft clauses to ensure optimality. In other words, it enumerates all satisfying assignments to the SAT encoding and then returns the best solution. This variant also uses minimal failing sub-contracts for pruning the search space.
- Solidare-Baseline neither uses soft clauses nor minimal failing sub-contracts. Instead, it uses the same hard constraints as Solidare and then adds $\neg M$ as a blocking clause in each iteration.

The results of this comparison are shown in Figure 16 where the x-axis denotes the number of solved sketches within a 20-minute time limit, and the y-axis represents synthesis time on *log-scale*. As we can see here, our proposed sketch completion algorithm significantly outperforms three baselines, which fail to solve 10% (for Solidare-NoSoft), 22% (for Solidare-NoMFS) and 24% (for Solidare-Baseline) of the benchmarks. In contrast, Solidare can solve all sketches with an average (resp. median) sketch solving time of 12.8 (resp. 0.9) seconds.

RQ4: Our MaxSAT-based sketch solver that utilizes minimal failing sub-contracts significantly outperforms the three simpler sketch completion baselines.

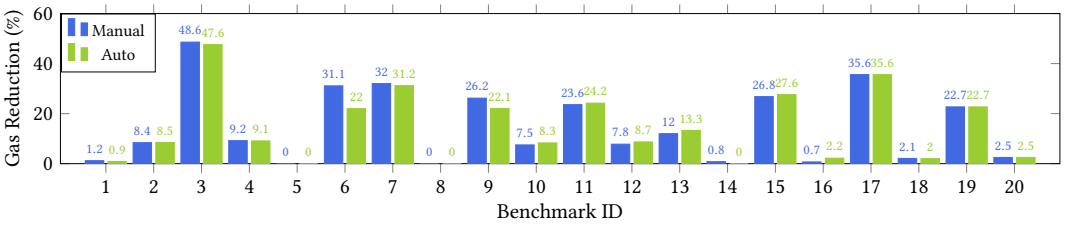

Fig. 17. Comparison of gas reduction between manual refactoring and autotuning. The autotuner does not find a gas-saving refactoring for benchmarks 5, 8, or 14.

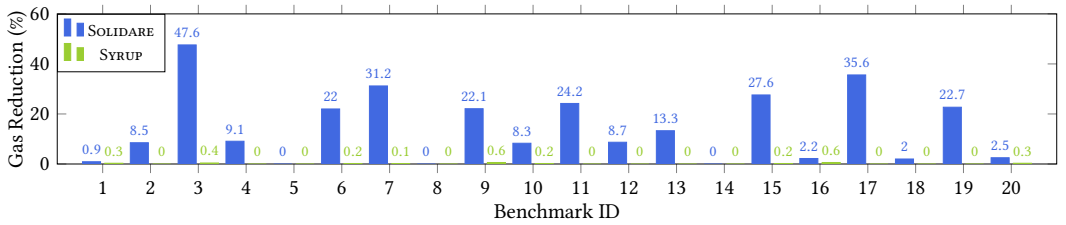

Fig. 18. Comparing Solidare and Syrup. Neither tool can optimize benchmark 5, 8, or 14.

7.5 Effectiveness of Auto-tuning

To answer RQ5, we also evaluate the auto-tuning component of Solidare on the 20 Solidity programs in the Etherscan dataset. Specifically, for each of these programs, we let the auto-tuner explore different data type refactorings for up to 10 minutes and then report the gas usage of the best refactoring encountered during this exploration process.

The results of this experiment are presented in Figure 17. Here, each bar corresponds to percentage gas usage improvement over the original version of the contract. In more detail, the green bars correspond to the refactoring discovered by the auto-tuner, and the blue bars are for the best transformation written by us. As we can see from Figure 17, the auto-tuner can automatically reduce gas usage for 17 out of 20 (85%) benchmarks. The reduction in gas usage is more than 5% for 13 out of 20 contracts (65%) and more than 10% for 9 (45%) . Furthermore, for 7 benchmarks, the refactoring returned by the auto-tuner is actually *better* than the best manually-written refactoring.

There are three benchmarks whose gas usage cannot be optimized at all using our auto-tuner (benchmarks 5, 8, and 14). Among these, Congress (benchmark 5) and DAOG2X (benchmark 8) can also not be improved using the transformations written by us, and, for Kingdom (benchmark 14), the improvement for the manually written transformation is very small. Overall, we believe these results indicate that Solidare's auto-tuner is fairly effective and that Solidare can be used to reduce gas usage automatically in many cases.

> **RQ5:** Using its auto-tuner component, Solidare is able to find gas-saving refactorings for 85% of the benchmarks in the Etherscan dataset. Furthermore, the average gas improvement for automatically found refactorings is competitive with manually written ones (17.0% for auto-tuner and 16.6% for manually written).

7.6 Comparison with Existing Gas Optimizer

To answer RQ6, we compare Solidare (in its fully automated mode using the auto-tuner) against a state-of-the-art tool called Syrup [Albert et al. 2020b] which is a super-optimizer that aims to

reduce gas usage of smart contracts. Since all smart contracts in the GasStation dataset use Solidity 0.8.0 and above, which is not supported by Syrup, we perform the comparison over the Etherscan dataset. The result is shown in Figure 18, where the blue and green bars indicate percentage gas reduction using Solidare and Syrup, respectively. As is evident from Figure 18, Syrup cannot reduce gas usage as much as Solidare on these benchmarks and results in an average gas saving of up to 0.6%.[12] In particular, Syrup focuses on optimizing arithmetic operations within a basic block, but more significant gas savings require changing the data layout, which Syrup does not handle. Nonetheless, we believe that the types of optimizations performed by Syrup are complementary to our proposed approach.

RQ6: Solidare complements existing Solidity optimizers by improving gas usage through data layout transformations. In particular, Syrup only achieves an average of 0.3% gas reduction on our 20 benchmarks in Etherscan, while Solidare achieves 16.9%.

8 RELATED WORK

Data representation synthesis. Our work is related to a line of research on data representation synthesis [Delaware et al. 2015; Feser et al. 2020; Hawkins et al. 2011, 2012; Loncaric et al. 2018, 2016; Pailoor et al. 2021; Qiu and Solar-Lezama 2017; Solar-Lezama et al. 2008], where the goal is to synthesize a data representation and its operations given a high-level specification or reference implementation. For example, RelC [Hawkins et al. 2011, 2012] synthesizes data structures using relational algebra and functional dependencies. Similarly, Cozy [Loncaric et al. 2018, 2016] synthesizes data structures from high-level specifications and supports a richer family of operations such as inequalities, aggregations of multiple related collections, etc. However, Cozy requires a full functional specification written in its custom DSL, whereas Solidare rewrites the original program. Besides data structures, prior work also synthesizes abstract data types [Delaware et al. 2015] and data layouts for relational storage [Feser et al. 2020]. In contrast, Solidare solves a different problem: automatically re-implementing a program based on user-provided type-level refactoring operations. In this sense, Solidare is mostly related to Prism [Curino et al. 2013] which aims to evolve database applications based on user-provided schema modification operators. It is also related to Migrator [Wang et al. 2019a] that synthesizes an equivalent database program over the new schema. However, the problem solved by Solidare is more challenging than those solved by Prism and Migrator: rather than re-writing SQL queries, Solidare needs to re-implement an entire program written in an imperative programming language.

Type-level refactorings. Our work is also related to a line of research on type-level refactorings [Balaban et al. 2005; Dig et al. 2008; Henkel and Diwan 2005; Ketkar et al. 2019; Steimann and Thies 2009; Tip et al. 2011, 2003], where the goal is to preserve program behavior after changing type names or class definitions. For example, T2R [Ketkar et al. 2019] solves the type migration problem that aims to replace an existing type T with another type R, such as replacing HashMap with TreeMap. It performs type constraint analysis to handle dependencies that propagate over assignment operations, method hierarchies, and subtypes. Another related work is that of Steimann et al. [Steimann and Thies 2009] which focuses on changes to access modifiers (e.g. from public to private). It formalizes the accessibility of fields and methods as constraint rules and computes necessary changes of access modifiers to preserve program behavior. However, none of these prior efforts can handle data type refactorings addressed in this paper.

[12] Syrup achieves similar gas savings of 0.59% on the benchmarks used in [Albert et al. 2020b].

Smart contract analysis. There have been many proposals for analyzing smart contracts, including fuzzing [Jiang et al. 2018], runtime validation [Li et al. 2020], symbolic execution [Kalra et al. 2018; Luu et al. 2016], type systems [Das et al. 2020], and formal verification [Lin et al. 2021; Permenev et al. 2020; Wang et al. 2019b]. Most techniques aim to find security vulnerabilities or prove safety. For example, Oyente [Luu et al. 2016] performs symbolic execution to identify vulnerabilities like reentrancy and transaction-order dependencies. Solythesis [Li et al. 2020] instruments a smart contract based on a user-specified invariant and produces a new contract that rejects all transactions violating the invariant. In contrast, Solidare aims to reduce gas usage but leverages an equivalence checker to establish correctness.

Gas optimization for smart contracts. Existing approaches for gas optimization can be categorized into two classes: superoptimization and pattern-based. Superoptimization approaches [Albert et al. 2020a,b] find the gas-optimal block of code in the CFG of the smart contract. For instance, Albert et al. [Albert et al. 2020b] use a Max-SMT solver to search for optimized versions of a contract, and Gasol [Albert et al. 2020a] optimizes a contract by replacing storage access with semantics-preserving memory access. On the other hand, pattern-based approaches [Chen et al. 2017, 2018] optimize contracts based on pre-defined rewrite rules. For example, GASPER [Chen et al. 2017] uses symbolic execution to identify seven gas-costly programming patterns such as dead code, repeated computations in a loop, etc. We believe these approaches are complementary to ours and can be used in conjunction with Solidare. Finally, it is worth noting that Solidare can also benefit from advances in smart contract gas analysis [Albert et al. 2020a, 2019; Das and Qadeer 2020; Sergey et al. 2019] to reject candidate programs that are guaranteed not to reduce gas usage.

9 CONCLUSION

We have presented Solidare, a new tool for automatically re-implementing smart contracts based on type-level refactorings provided by the developer. Solidare provides a DSL for expressing type-level refactorings, and, given a Solidity program and a type-level transformation expressed in this DSL, Solidare automatically synthesizes an equivalent Solidity program using the new data types. Our experiments on a set of real-world contracts demonstrate that Solidare is practical and can be used to reduce gas usage. Furthermore, when used with the proposed auto-tuner, Solidare can be used to reduce the gas usage of smart contracts fully automatically.

ACKNOWLEDGMENTS

We would like to thank Shankara Pailoor, Rong Pan, and the anonymous reviewers for their helpful comments on an earlier version of this paper. This material is based on research sponsored by NSF CCF-1811865, CCF-1712067, SaTC-1908494, NSERC Discovery Grant, DARPA N66001-22-C-4028, and Google Faculty Research Award.

REFERENCES

Elvira Albert, Jesús Correas, Pablo Gordillo, Guillermo Román-Díez, and Albert Rubio. 2020a. Gasol: Gas analysis and optimization for ethereum smart contracts. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 118–125.

Elvira Albert, Pablo Gordillo, Albert Rubio, and Maria Anna Schett. 2020b. Synthesis of Super-Optimized Smart Contracts Using Max-SMT. (2020), 177–200.

Elvira Albert, Pablo Gordillo, Albert Rubio, and Ilya Sergey. 2019. Running on Fumes - Preventing Out-of-Gas Vulnerabilities in Ethereum Smart Contracts Using Static Resource Analysis. In *Proceedings of International Conference on Verification and Evaluation of Computer and Communication Systems (VECoS)*. Springer, 63–78.

Ittai Balaban, Frank Tip, and Robert M. Fuhrer. 2005. Refactoring support for class library migration. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming (OOPSLA)*. ACM, 265–279.

Vitalik Buterin. 2014. Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform. https://ethereum.org/en/whitepaper.

Ting Chen, Xiaoqi Li, Xiapu Luo, and Xiaosong Zhang. 2017. Under-optimized smart contracts devour your money. In *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE Computer Society, 442–446.

Ting Chen, Zihao Li, Hao Zhou, Jiachi Chen, Xiapu Luo, Xiaoqi Li, and Xiaosong Zhang. 2018. Towards saving money in using smart contracts. In *2018 IEEE/ACM 40th International Conference on Software Engineering: New Ideas and Emerging Technologies Results (ICSE-NIER)*. IEEE, 81–84.

Carlo Curino, Hyun Jin Moon, Alin Deutsch, and Carlo Zaniolo. 2013. Automating the database schema evolution process. *VLDB J.* 22, 1 (2013), 73–98.

Ankush Das, Stephanie Balzer, Jan Hoffmann, Frank Pfenning, and Ishani Santurkar. 2020. Language-Based Web Session Integrity. In *Proceedings of the Computer Security Foundations Symposium (CSF)*. IEEE, 107–122.

Ankush Das and Shaz Qadeer. 2020. Exact and Linear-Time Gas-Cost Analysis. 12389 (2020), 333–356.

Benjamin Delaware, Clément Pit-Claudel, Jason Gross, and Adam Chlipala. 2015. Fiat: Deductive Synthesis of Abstract Data Types in a Proof Assistant. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 689–700.

Danny Dig, Stas Negara, Vibhu Mohindra, and Ralph E. Johnson. 2008. *ReBA*: *re*factoring-aware *b*inary *a*daptation of evolving libraries. In *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, 441–450.

Ethereum. 2022. Ethereum Virtual Machine. https://ethereum.org/en/developers/docs/evm.

Etherscan. 2022. https://etherscan.io.

ETHGasStation. 2022. https://ethgasstation.info.

Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. 2018. Program synthesis using conflict-driven learning. (2018), 420–435.

John Feser, Sam Madden, Nan Tang, and Armando Solar-Lezama. 2020. Deductive optimization of relational data storage. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–30.

Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin C. Rinard, and Mooly Sagiv. 2011. Data representation synthesis. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 38–49.

Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin C. Rinard, and Mooly Sagiv. 2012. Concurrent data representation synthesis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, (PLDI)*. ACM, 417–428.

Johannes Henkel and Amer Diwan. 2005. CatchUp!: capturing and replaying refactorings to support API evolution. In *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, 274–283.

Bo Jiang, Ye Liu, and W. K. Chan. 2018. ContractFuzzer: fuzzing smart contracts for vulnerability detection. In *Proceedings of the ACM/IEEE International Conference on Automated Software Engineering, (ASE)*. ACM, 259–269.

Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. ZEUS: Analyzing Safety of Smart Contracts. In *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS)*. The Internet Society.

Ameya Ketkar, Ali Mesbah, Davood Mazinanian, Danny Dig, and Edward Aftandilian. 2019. Type migration in ultra-large-scale codebases. In *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE / ACM, 1142–1153.

Daniel Le Berre and Anne Parrain. 2010. The sat4j library, release 2.2, system description. *Journal on Satisfiability, Boolean Modeling and Computation* 7 (2010), 59–64.

Ao Li, Jemin Andrew Choi, and Fan Long. 2020. Securing smart contract with runtime validation. In *Proceedings of the ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 438–453.

Chu Min Li and Felip Manya. 2009. MaxSAT, Hard and Soft Constraints. *Handbook of satisfiability* 185 (2009), 613–631.

Shaokai Lin, Xinyuan Sun, Jianan Yao, and Ronghui Gu. 2021. SciviK: A Versatile Framework for Specifying and Verifying Smart Contracts. *CoRR* abs/2103.02209 (2021). arXiv:2103.02209 https://arxiv.org/abs/2103.02209

Calvin Loncaric, Michael D. Ernst, and Emina Torlak. 2018. Generalized data structure synthesis. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*. ACM, 958–968.

Calvin Loncaric, Emina Torlak, and Michael D. Ernst. 2016. Fast synthesis of fast collections. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 355–368.

Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 254–269.

Inês Lynce and Joao P Marques-Silva. 2004. On computing minimum unsatisfiable cores. (2004).

Benjamin Mariano, Yanju Chen, Yu Feng, Shuvendu K. Lahiri, and Isil Dillig. 2020. Demystifying Loops in Smart Contracts. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*. IEEE, 262–274.

Joao Marques-Silva, Inês Lynce, and Sharad Malik. 2009. Conflict-driven clause learning SAT solvers. In *Handbook of satisfiability*. ios Press, 131–153.

John C Mitchell. 1991. On the equivalence of data representations. *Artificial intelligence and mathematical theory of computation: papers in honor of John McCarthy* (1991), 305–330.

Julian Nagele and Maria A Schett. 2020. Blockchain Superoptimizer. *arXiv preprint arXiv:2005.05912* (2020).

Shankara Pailoor, Yuepeng Wang, Xinyu Wang, and Isil Dillig. 2021. Synthesizing data structure refinements from integrity constraints. In *Proceedings of the ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*. ACM, 574–587.

Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin T. Vechev. 2020. VerX: Safety Verification of Smart Contracts. In *IEEE Symposium on Security and Privacy (SP)*. IEEE, 1661–1677.

Xiaokang Qiu and Armando Solar-Lezama. 2017. Natural synthesis of provably-correct data-structure manipulations. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 65:1–65:28.

Ilya Sergey, Vaivaswatha Nagaraj, Jacob Johannsen, Amrit Kumar, Anton Trunov, and Ken Chan Guan Hao. 2019. Safer smart contract programming with Scilla. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 185:1–185:30.

Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodík. 2008. Sketching concurrent data structures. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 136–148.

Friedrich Steimann and Andreas Thies. 2009. From Public to Private to Absent: Refactoring Java Programs under Constrained Accessibility. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP) (Lecture Notes in Computer Science, Vol. 5653)*. Springer, 419–443.

Frank Tip, Robert M. Fuhrer, Adam Kiezun, Michael D. Ernst, Ittai Balaban, and Bjorn De Sutter. 2011. Refactoring using type constraints. *ACM Trans. Program. Lang. Syst.* 33, 3 (2011), 9:1–9:47.

Frank Tip, Adam Kiezun, and Dirk Bäumer. 2003. Refactoring for generalization using type constraints. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems (OOPSLA)*. ACM, 13–26.

Yuepeng Wang, James Dong, Rushi Shah, and Isil Dillig. 2019a. Synthesizing database programs for schema refactoring. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 286–300.

Yuepeng Wang, Shuvendu K. Lahiri, Shuo Chen, Rong Pan, Isil Dillig, Cody Born, Immad Naseer, and Kostas Ferles. 2019b. Formal Verification of Workflow Policies for Smart Contracts in Azure Blockchain. In *Proceedings of the International Conference on Verified Software. Theories, Tools, and Experiments (VSTTE) (Lecture Notes in Computer Science, Vol. 12031)*. Springer, 87–106.

Yuepeng Wang, Rushi Shah, Abby Criswell, Rong Pan, and Isil Dillig. 2020. Data Migration using Datalog Program Synthesis. *Proceedings of the VLDB Endowment* 13, 7 (2020), 1006–1019.

Gavin Wood. 2022. Ethereum: A Secure Decentralised Generalised Transaction Ledger. https://ethereum.github.io/yellowpaper/paper.pdf.