Populating the Peephole Optimizer of a Smart Contract Compiler

Maria A. Schett
University College London, UK
http://maria-a-schett.net
mail@maria-a-schett.net

Julian Nagele
London, UK
http://jnagele.net
mail@jnagele.net

Abstract

Developing compiler optimizations, especially for new, rapidly evolving smart contract languages, can be onerous and error-prone, but is especially important for smart contracts, where deployment and execution directly translate to monetary cost and which cannot change once deployed. One common optimization technique is the use of *peephole optimizations*, replacement rules that are applied using pattern-matching. These rules are normally constructed using human expertise, which is both time-consuming and far from systematic in exploring opportunities for optimization. In this work we propose a pipeline to automatically populate the peephole optimizer of a smart contract compiler. We apply superoptimization to an existing code base to obtain sequences of instructions, which can be replaced by cheaper, observationally equivalent instructions. We then generate peephole optimization rules by extracting the underlying patterns of these optimizations. We provide a case study of our approach and a prototype implementation for bytecode of the Ethereum Virtual Machine, the tool ppltr, which combines the superoptimizer ebso and the rule generator sorg. Then we evaluate our approach by generating and applying nearly $1k$ peephole optimization rules extracted from $2k$ optimizations obtained from deployed bytecode.

2012 ACM Subject Classification Software and its engineering → Formal methods; Software and its engineering → Compilers

Keywords and phrases Compiler Optimizations, Constraint Solving, Ethereum Bytecode

Digital Object Identifier 10.4230/OASIcs.FMBC.2020.3

Supplementary Material https://github.com/mariaschett/ppltr

1 Introduction

In this work we leverage formal methods to automatically populate the peephole optimizer of a smart contract compiler. A *peephole optimizer* uses pattern matching to optimize a small fragment of code, i.e., a peephole, by applying *optimization rules*. But finding sound optimization rules is a bottleneck as witnessed by the peephole optimizer of the Solidity compiler solc.[1] Currently, solc features fewer than 20 rules compared to LLVM's 1000+ rules. Thus we propose a pipeline to automatically populate the peephole optimizer of a smart contract compiler by combining techniques from constraint solving and rewriting as illustrated in Figure 1.

Smart contract languages typically have a large and accessible code base to use as a basis for finding optimizations, e.g., code deployed to public blockchains or test cases.

[1] github.com/ethereum/solidity/blob/ 019ec63f63bae7bbe89f5b62bb7b202ef5dadce6/libevmasm/PeepholeOptimiser.cpp

2nd Workshop on Formal Methods for Blockchains (FMBC 2020).
Editors: Bruno Bernardo and Diego Marmsoler; Article No. 3; pp. 3:1–3:15
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

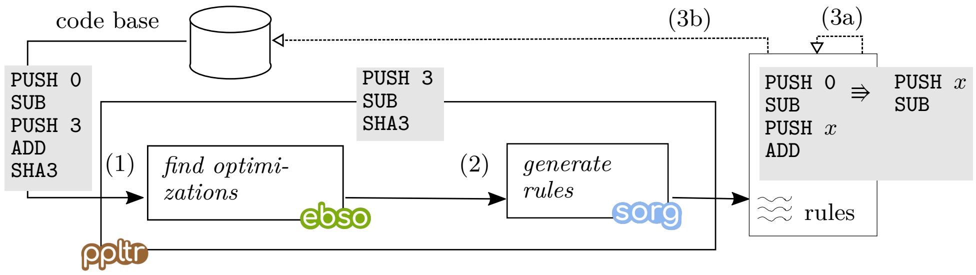

Figure 1 Pipeline to automatically generate peephole optimization rules from a code base.

(1) This allows us to start from an existing code base, to *find optimizations* by using automated tools to synthesize observationally equivalent but cheaper instruction sequences.

This automatic synthesis is possible, because many smart contract languages come with formally defined operational semantics, e.g., the Ethereum yellow paper [22]. Moreover, execution of a smart contract comes with a clear cost model – gas – giving rise to a precise notion of optimality. To give an example, the bytecode for the Ethereum virtual machine `PUSH 0 SUB PUSH 3 ADD SHA3` computes a hash of $3 + (0 - w)$ for some word w already on the stack. As $3 + (0 - w) = 3 - w$ the bytecode corresponding to `PUSH 3 SUB SHA3`, computes the same result *and* cheaper.

(2) From such optimizations, we can *generate rules*. Using concepts from rewriting we generalize "unnecessarily specific" arguments and strip away "unnecessary" context to obtain optimization rules.

For the above example, we generate the rule `PUSH 0 SUB PUSH` x `ADD` $\Rightarrow$ `PUSH` x `SUB` by generalizing 3 to x.

(3) Finally we can feed back and apply the generated rules to

- **(a)** the rules themselves, and
- **(b)** the code base and again start the cycle to find new optimizations.

We demonstrate the applicability of our pipeline in a case study for bytecode of the Ethereum virtual machine (EVM). We implemented a prototype: ppltr, a peephole otimization rule generator. For phase (1), we use the tool ebso [19], a superoptimizer for EVM bytecode. For phase (2), we use sorg, a superoptimization based rule generator. All tools are available open-source under the Apache-2.0 license.[2] We evaluated our approach on bytecode of the 250 most called contracts of the Ethereum blockchain, where we found 2032 distinct optimizations from which we automatically generated 993 optimization rules.

Contributions

1. We propose a pipeline for automatically populating a peephole optimizer, and
2. a sound and complete procedure to generate optimization rules from optimizations.
3. We perform a case study for EVM bytecode with
4. a prototype implementation, together with
5. an evaluation.

[2] Available at `github.com/juliannagele/ebso/tree/v2.1`, `github.com/mariaschett/sorg/tree/v1.1`, and `github.com/mariaschett/ppltr/tree/v1.0`.

2 Approach

We assume a machine model with a *state* over a set of *words* $\mathbb{W}$ with an observational equivalence relation $\equiv$ on states, which may take only parts of the state into account. States are modified based on *instructions* from a set $\mathcal{I}$, where an instruction $\iota \in \mathcal{I}$ deterministically transforms a state σ into some state σ' denoted by $\sigma \xrightarrow{\iota} \sigma'$. Some instructions act only on parts of the state, while others take immediate arguments from $\mathbb{W}$. We write $\iota(w_1, \dots, w_k)$ for an instruction $\iota \in \mathcal{I}$ which takes k immediate arguments $w_1, \dots, w_k \in \mathbb{W}$ and say that ι has arity k. For example, in a stack-based machine the instruction `PUSH 3` takes the immediate argument 3, while `SUB` has arity 0, but consumes two arguments from the stack.

A *program* ρ is a sequence of instructions $\iota_0 \cdots \iota_n$. The length of ρ is its number of instructions, denoted by $|\rho|$. We write ε for the *empty* program and $\rho \cdot \tau$ for the concatenation of programs ρ and τ. A program $\rho = \iota_0 \cdots \iota_n$ transforms a family of states $\boldsymbol{\sigma} = (\sigma_j)_{j \leq n+1}$ by stepwise transformation, i.e., $\sigma_0 \xrightarrow{\iota_0} \sigma_1 \xrightarrow{\iota_1} \dots \xrightarrow{\iota_n} \sigma_{n+1}$, and we write $\sigma_0 \overset{\rho}{\twoheadrightarrow} \sigma_{n+1}$. Here σ_j is the state after executing j instructions, and σ_0 is the designated start state. We often write states instead of families of states, when the distinction is clear from the context.

We write $\mathsf{cost}(\iota, \sigma)$ for the cost incurred by executing instruction ι on state σ. The cost of executing a program is simply the sum of the cost of its instructions: $\mathsf{cost}(\iota_0 \cdots \iota_n, \boldsymbol{\sigma}) = \sum_{j=0}^{n} \mathsf{cost}(\iota_j, \sigma_j)$. Two programs ρ and τ are *equal*, denoted by $\rho = \tau$, if they are syntactically equal, and *equivalent*, $\rho \equiv \tau$, if they are observationally equivalent, i.e., for states $\boldsymbol{\sigma}$ and $\boldsymbol{\sigma'}$ with $\sigma_0 \equiv \sigma'_0$, $\sigma_0 \overset{\rho}{\twoheadrightarrow} \sigma_{|\rho|+1}$, and $\sigma'_0 \overset{\tau}{\twoheadrightarrow} \sigma'_{|\tau|+1}$ we have $\sigma_{|\rho|+1} \equiv \sigma'_{|\tau|+1}$.

▶ **Definition 1.** *Let ρ and τ be programs with $\rho \equiv \tau$ and $\mathsf{cost}(\rho, \boldsymbol{\sigma}) > \mathsf{cost}(\tau, \boldsymbol{\sigma})$ for all states $\boldsymbol{\sigma}$. Then τ is an* optimization *of ρ, and we write $\rho \gtrapprox \tau$.*

In Section 2.1, we will show how we can obtain such optimizations – and in Section 2.2 we will use them to generate optimization rules. To do so, we need to define what constitutes a rule. Therefore we abstract over the immediate arguments of instructions by using a countably infinite set of variables $\mathcal{V}$. We extend $\mathcal{I}$ to $\mathcal{I}^{\mathcal{V}}$ by adding instructions $\iota(x_1, \dots, x_k)$ for all $x_1, \dots, x_k \in \mathcal{V}$ and all $\iota \in \mathcal{I}$ of arity $k > 0$.

A program over $\mathcal{I}^{\mathcal{V}}$ is called a *program schema*. To obtain a *maximal schema* of a program schema s every $\iota(w_1, \dots, w_k)$ in s is replaced by $\iota(x_1, \dots, x_k)$, where $x_1, \dots, x_k$ are fresh variables from $\mathcal{V}$. All variables in a program schema s are collected in $\mathcal{V}\mathsf{ar}(s)$.

A *substitution* $\gamma : \mathcal{V} \to \mathbb{W} \cup \mathcal{V}$ maps variables to variables and words. In a *ground* substitution $\overline{\gamma}$ the range is restricted to $\mathbb{W}$, i.e., $\overline{\gamma} : \mathcal{V} \to \mathbb{W}$. We *apply* γ to a schema s by replacing all variables x in s by $\gamma(x)$ and write $s\gamma$ for the result. Note that $s\overline{\gamma}$ is a program. A substitution γ is *at least as general* as a substitution γ', denoted $\gamma \leq \gamma'$, if there is a substitution γ'' such that $\gamma\gamma'' = \gamma'$. If $\gamma \leq \gamma'$ and $\gamma' \nleq \gamma$ then we say γ is *more general* than γ' and write $\gamma < \gamma'$.

We call program schemas s and t *observationally equivalent*, and write $s \equiv t$, if $s\overline{\gamma} \equiv t\overline{\gamma}$ holds for all $\overline{\gamma}$ and write $\mathsf{cost}(s, \boldsymbol{\sigma}) > \mathsf{cost}(t, \boldsymbol{\sigma'})$ if $\mathsf{cost}(s\overline{\gamma}, \boldsymbol{\sigma}) > \mathsf{cost}(t\overline{\gamma}, \boldsymbol{\sigma'})$ for all $\overline{\gamma}$.

▶ **Definition 2.** *Let ℓ and r be program schemas with $\ell \equiv r$ and $\mathsf{cost}(\ell, \boldsymbol{\sigma}) > \mathsf{cost}(r, \boldsymbol{\sigma})$. Then $\ell \Rrightarrow r$ is an (optimization)* rule.

By definition, every optimization $\rho \gtrapprox \tau$ is an optimization rule $\rho \Rrightarrow \tau$. A *context* C is a pair of program schemas (s_1, s_2). We write $C[t]$ for the program schema $s_1 \cdot t \cdot s_2$ and call s_1 a *prefix* and s_2 a *postfix* of $C[t]$. A context (s_1, s_2) is *at least as general* as a context (t_1, t_2), denoted by $(s_1, s_2) \leq (t_1, t_2)$, if there is a context (r_1, r_2) such that $r_1 \cdot s_1 = t_1$ and $s_2 \cdot r_2 = t_2$. If $C \leq C'$ and $C' \nleq C$ then we say C is *more general* than C' and write $C < C'$.

The following definition captures all optimization rules that can produce a given optimization when instantiated.

▶ **Definition 3.** *The* optimization rules *for an optimization $\rho \gtreqless \tau$ are defined as $\mathcal{R}(\rho \gtreqless \tau) = \{\ell \Rrightarrow r \mid \rho = C[\ell\gamma]$ and $\tau = C[r\gamma]$ for some substitution γ and context $C\}$.*

We ensure that applying peephole optimizations is sound by the following lemma.

▶ **Lemma 4.** *If $\rho \equiv \tau$ then $C[\rho] \equiv C[\tau]$ for all contexts C.*

Proof. We show the statement by induction on C. By assumption, the statement holds for the base case $C = (\varepsilon, \varepsilon)$. For the step case $C = (\iota \cdot s_1, s_2)$ observe that every instruction ι is deterministic, i.e., executing ι starting from a state σ leads to a deterministic state σ'. By induction hypothesis, executing $s_1 \rho s_2$ and $s_1 \tau s_2$ from a state σ' leads to an observationally equivalent state σ'', and therefore $\iota \cdot s_1 \cdot \rho \cdot s_2 \equiv \iota \cdot s_1 \cdot \tau \cdot s_2$ holds. We can reason analogously for $C = (s_1, s_2 \cdot \iota)$. ◀

2.1 Find Optimizations

As Definition 1 suggests finding an optimization for a program ρ necessitates finding

1. an observationally equivalent program τ, where
2. the cost of τ is less than the cost of ρ.

We leverage a constraint solver, such as Z3 [8], to automatically find equivalent, but cheaper programs. To this end, we express the above as an SMT problem: given a source program ρ, is there a target program τ such that for all possible inputs, executing ρ and τ results in the same final state, but the cost of τ is less than the cost of ρ? Our encoding is based on the encoding from unbounded superoptimization [11].

Find an Observationally Equivalent Program

To encode observational equivalence we first need a constraint that expresses equality on states: Let $\mathsf{enc\_eq\_state}(\sigma, \sigma')$ be an SMT constraint that evaluates to true, whenever state σ and state σ' are observationally equivalent. The concrete instantiation of this constraint depends on the machine that is modeled. For instance, the state may be modeled as several uninterpreted functions. An encoding for the EVM, modeling the state with a stack, storage, and exceptional halting can be found in Example 16, with the corresponding encoding of enc_eq_state in Example 18.

Based on the operational semantics for every $\iota \in \mathcal{I}$, we need to encode the effect of ι on a state i.e., the relation $\xrightarrow{\iota}$.

▶ **Definition 5.** *Let $\mathsf{enc\_step}(\iota, \sigma, \sigma')$ be an SMT encoding of the effect of an instruction ι as constraints between state σ and state σ'. For a program $\rho = \iota_0 \cdots \iota_n$ and states $\boldsymbol{\sigma}$ we define $\mathsf{enc\_progr}(\rho, \boldsymbol{\sigma})$ as $\bigwedge_{0 \leqslant j \leqslant n} \mathsf{enc\_step}(\iota_j, \sigma_j, \sigma_{j+1})$.*

Again, the concrete encoding of enc_step depends on the machine that is modeled, see Example 17 for our instantiation for the EVM.

Most programs will consume some input words $\vec{x}$. To pass them to the program, we assume an encoding $\mathsf{enc\_init}(\vec{x}, \boldsymbol{\sigma})$ that sets constraints on the start state σ_0 appropriately, e.g., putting the words in $\vec{x}$ in registers or on the stack according to the machine model. Based on the constraint enc_step, we can encode the search space of all possible target programs. To this end we represent the target program as a pair $\tau = \langle \mathsf{instr}, n \rangle$ of an uninterpreted function $\mathsf{instr}(j) : \mathbb{N} \to \mathcal{I}$ and its length $n \in \mathbb{N}$. The function instr acts as a template to be filled by the SMT solver returning the instruction to be used at position j of the target program. After a model has been found, the concrete target program can be reconstructed as $\mathsf{instr}(0) \cdot \mathsf{instr}(1) \cdots \mathsf{instr}(n-1)$.

▶ **Definition 6.** *Given a set of instructions $\mathcal{I}$ we define the SMT encoding for the enumeration of every program of length n as* $\mathsf{enc\_search}(\tau, \boldsymbol{\sigma})$ *as*

$$\forall j.\, 0 \leqslant j < n \rightarrow \bigwedge_{\iota \in \mathcal{I}} \mathsf{instr}(j) = \iota \rightarrow \mathsf{enc\_step}(\iota, \sigma_j, \sigma_{j+1}) \wedge \bigvee_{\iota \in \mathcal{I}} \mathsf{instr}(j) = \iota \tag{1}$$

The first clause states that if we pick ι at position j, then the effect is determined by $\mathsf{enc\_step}(\iota, \sigma_j, \sigma_{j+1})$. The second clause, $\bigvee_{\iota \in \mathcal{I}} \mathsf{instr}(j) = \iota$, ascertains that for every position j some instruction is picked.

▶ **Definition 7.** *The encoding for finding an observationally equivalent program to a given program ρ is*

$$\begin{aligned} &\exists n, \forall \vec{x}.\, \mathsf{enc\_init}(\vec{x}, \boldsymbol{\sigma}) \wedge \mathsf{enc\_init}(\vec{x}, \boldsymbol{\sigma'}) \wedge \\ &\qquad \mathsf{enc\_progr}(\rho, \boldsymbol{\sigma}) \wedge \mathsf{enc\_search}(\tau, \boldsymbol{\sigma'}) \wedge \mathsf{enc\_eq\_state}(\sigma_{|\rho|+1}, \sigma'_n) \end{aligned} \tag{2}$$

The first two constraints initialize states $\boldsymbol{\sigma}$ and $\boldsymbol{\sigma'}$ with the same inputs, the third and fourth constraint encode the effects of the existing program ρ and the sought after target program τ respectively, while the final constraint demands that they are observationally equivalent, i.e., that they result in equivalent states. With this constraint we will find observational equivalent programs. Now we will need to add constraints on the cost.

Find a Cheaper Program

To achieve this we extend Constraint (2) from Definition 7 by a constraint stating that the cost of executing the target program τ is less than the cost of executing the source program ρ: i.e., $\mathsf{cost}(\rho, \boldsymbol{\sigma}) > \mathsf{cost}(\tau, \boldsymbol{\sigma'})$. Here the cost of τ is again defined by summation, i.e., for $\tau = \langle \mathsf{instr}, n \rangle$ we have $\mathsf{cost}(\tau, \boldsymbol{\sigma'}) = \sum_{j=0}^{n-1} \mathsf{cost}(\mathsf{instr}(j), \sigma_j)$.

2.2 Generate Rules

As Definition 3 indicates generating optimization rules from optimizations requires us

1. to find a substitution γ, and
2. to find a context C.

Find a Substitution

In the first step we generalize the immediate arguments of instructions in an optimization $\rho \gtreqless \tau$ by finding a substitution. We capture all possible generalizations of a rule using the following definition.

▶ **Definition 8.** *The* generalized rules *of an optimization rule $\rho \Rrightarrow \tau$ are defined as $\mathcal{G}(\rho \Rrightarrow \tau) = \{\ell \Rrightarrow r \mid \ell\gamma = \rho$ and $r\gamma = \tau$ for some substitution $\gamma\}$.*

▶ **Example 9.** Let $\rho \equiv \tau$ be the optimization from the introduction, i.e., `PUSH 0 SUB PUSH 3 ADD SHA3` $\equiv$ `PUSH 3 SUB SHA3`. Then $\mathcal{G}(\rho \equiv \tau)$ consists of two rules: `PUSH 0 SUB PUSH` x `ADD SHA3` $\Rrightarrow$ `PUSH` x `SUB SHA3` and $\rho \Rrightarrow \tau$ itself. Note that the pair `PUSH` y `SUB PUSH` x `ADD SHA3` and `PUSH` x `SUB SHA3` is not in $\mathcal{G}(\rho \equiv \tau)$. Applying the substitution $\gamma = \{x \mapsto 3, y \mapsto 0\}$ would yield the original optimization, but since `PUSH` y `SUB PUSH` x `ADD SHA3` $\not\equiv$ `PUSH` x `SUB SHA3` they do not constitute an optimization rule.

Example 19 shows further generalized rules for EVM bytecode.

To implement $\mathcal{G}$ we can do an exhaustive search as follows: start from a maximal schema for the given optimization and try all possibilities of mapping the variables back to the original values, checking whether the result yields a rule. The following procedure implements this approach, additionally using an order on the candidate substitutions to prune the search space.

▶ **Definition 10.** *We define the function* generalize *as follows:*

```
 1: function generalize($(\rho \Rrightarrow \tau)$)
 2:     $\mathcal{R} \leftarrow \varnothing$
 3:     $\ell_0, r_0 \leftarrow$ maximal program schemas $\ell_o$ and $r_0$ for $\rho$ and $\tau$ with $\mathcal{V}\mathsf{ar}(\ell_0) \cap \mathcal{V}\mathsf{ar}(r_0) = \varnothing$
 4:     $\gamma_0 \leftarrow$ the substitution $\gamma_0$ with $\rho = \ell_0\gamma_0$ and $\tau = r_0\gamma_0$
 5:     $\Gamma \leftarrow \{\gamma \mid \gamma(x) = \gamma_0(x)$ or $\gamma(x) = y$ for $\gamma_0(x) = \gamma_0(y)$ and $x, y \in \mathcal{V}\mathsf{ar}(\ell_0) \cup \mathcal{V}\mathsf{ar}(r_0)\}$
 6:     for all $\gamma \in \Gamma$ do
 7:         if $\ell_0\gamma \equiv r_0\gamma$ then
 8:             $\mathcal{R} \leftarrow \mathcal{R} \cup \{\ell_0\gamma \Rrightarrow r_0\gamma\}$
 9:             $\Gamma \leftarrow \Gamma \setminus \{\gamma' \mid \gamma \lessdot \gamma'\}$
10:         else
11:             $\Gamma \leftarrow \Gamma \setminus \{\gamma' \mid \gamma' \lessdot \gamma\}$
12:     return $\mathcal{R}$
```

Using the order $\lessdot$ on substitutions to prune the search space is key for implementation. Pruning only removes rules covered by others as the following lemma shows.

▶ **Lemma 11.** *For every $\ell \Rrightarrow r \in \mathcal{G}(\alpha)$ of a rule α there is a $\ell' \Rrightarrow r' \in$* generalize*$(\alpha)$ and a substitution γ such that $\ell'\gamma = \ell$ and $r'\gamma = r$.*

Proof. We fix $\ell \Rrightarrow r \in \mathcal{G}(\alpha)$. Let ℓ_0 and r_0 be the maximal schemas of α. By definition of maximal schema there is a γ' such that $\ell_0\gamma' = \ell$ and $r_0\gamma' = r$. A renaming of γ' is in Γ and thus either generalize(α) will consider it at some point, or it will be removed by either line 9 or line 11.

If it is considered then a renaming of $\ell \Rrightarrow r$ is in generalize(α). If it is removed by line 9, then a substitution γ with $\gamma \lessdot \gamma'$ and and $\ell_0\gamma \equiv r_0\gamma$ was considered. Thus $\ell_0\gamma \Rrightarrow r_0\gamma$ is in generalize(α) and we have $\ell_0\gamma\gamma'' = \ell$ and $r_0\gamma\gamma'' = r$ for some γ'' by $\gamma \lessdot \gamma'$. If γ' was removed by line 11 then a substitution γ with $\gamma' \lessdot \gamma$ and and $\ell_0\gamma \not\equiv r_0\gamma$ was considered. But this contradicts the assumption $\ell \Rrightarrow r \in \mathcal{G}(\alpha)$, because observational equivalence is closed under substitution. ◀

Find a Context

As a second step We strip the generalized rules of any unnecessary pre- and postfix. Again we first capture all possible stripped rules and then give an implementation.

▶ **Definition 12.** *The* stripped rules *of a rule $\rho \Rrightarrow \tau$ are defined as $\mathcal{C}(\rho \Rrightarrow \tau) = \{\ell \Rrightarrow r \mid \rho = C[\ell]$ and $\tau = C[r]\}$.*

▶ **Example 13.** Continuing Example 9, for the rule `PUSH 0 SUB PUSH` x `ADD SHA3` $\Rrightarrow$ `PUSH` x `SUB SHA3` the stripped rules $\mathcal{C}$ contain the rule `PUSH 0 SUB PUSH` x `ADD` $\Rrightarrow$ `PUSH` x `SUB`, obtained by stripping away the context (ϵ, `SHA3`), and the original rule itself, since applying the empty context (ϵ, ϵ) to a program yields the program itself.

Example 20 shows further rules stripped of their context in EVM bytecode.

To implement $\mathcal{C}$ we follow the same strategy as for $\mathcal{G}$: try all possible contexts in an exhaustive search, checking whether they yield a rule and use an order contexts to prune the search space.

▶ **Definition 14.** *We define the function* strip *as*

```
1:  function strip($\rho \Rrightarrow \tau$)
2:      $\mathcal{R} \leftarrow \varnothing$
3:      $(s_0, t_0) \leftarrow$ the longest common prefix $s_0$ and the longest common postfix $t_0$ of $\rho$ and $\tau$
4:      $\ell_0, r_0 \leftarrow$ the program schemas $\ell_0$ and $r_0$ with $s_0 \cdot \ell_0 \cdot t_0 = \rho$ and $s_0 \cdot r_0 \cdot t_0 = \tau$
5:      $\Gamma \leftarrow \{C \mid C = (s,t)$ where $s' \cdot s = s_0$ and $t \cdot t' = t_0$ for some $s', t'\}$
6:      for all $C \in \Gamma$ do
7:          if $C[\ell_0] \equiv C[r_0]$ then
8:              $\mathcal{R} \leftarrow \mathcal{R} \cup \{C[\ell_0] \Rrightarrow C[r_0]\}$
9:              $\Gamma \leftarrow \Gamma \setminus \{C' \mid C < C'\}$
10:         else
11:             $\Gamma \leftarrow \Gamma \setminus \{C' \mid C' < C\}$
12:     return $\mathcal{R}$
```

Again, the order on contexts allows us to prune the search space without loss.

▶ **Lemma 15.** *For every $\ell \Rrightarrow r \in \mathcal{C}(\alpha)$ of a rule α there is a $\ell' \Rrightarrow r' \in$* strip$(\alpha)$ *and a context C such that $C[\ell'] = \ell$ and $C[r'] = r$.*

Proof. We fix a rule $\ell \Rrightarrow r \in \mathcal{C}(\alpha)$. Let (s_0, t_0) be the longest common prefix and the longest common postfix of α and be ℓ_0, r_0 the program schemas with $s_0 \cdot \ell_0 \cdot t_0 \Rrightarrow s_0 \cdot r_0 \cdot t_0 = \alpha$. A context C' with $C'[\ell_0] = \ell$ and $C'[r_0] = r$ is in Γ and thus either strip(α) will consider it at some point, or it will be removed by either line 9 or line 11.

If it is considered then $\ell \Rrightarrow r$ is in strip(α). If it is removed by line 9, then a context C with $C < C'$ and and $C[\ell_0] \equiv C[r_0]$ was considered. Thus $C[\ell_0] \Rrightarrow C[r_0]$ is in strip(α) and we have $C''[C[\ell_0] = \ell$ and $C''[C[r_0]] = r$ for some C'' by $C < C'$. If C' was removed by line 11 then a context C with $C' < C$ and and $C[\ell_0] \not\equiv C[r_0]$ was considered. Again this contradicts the assumption $\ell \Rrightarrow r \in \mathcal{C}(\alpha)$, because observational equivalence is closed under context. ◀

Soundness and Completeness

Finally, we combine the two functions and for an optimization $\rho \geqslant \tau$ define sorg$(\rho \geqslant \tau) = \{$strip$(\ell \Rrightarrow r) \mid \ell \Rrightarrow r \in$ generalize$(\rho \Rrightarrow \tau)\}$.

The rules generated by sorg$(\rho \geqslant \tau)$ are *sound*: for every $\ell \Rrightarrow r \in$ sorg$(\rho \Rrightarrow \tau)$ there is a substitution γ and a context C such that $C[\ell\gamma] = \rho$ and $C[r\gamma] = \tau$. This directly follows from generalize$(\rho \Rrightarrow \tau) \subseteq \mathcal{G}(\rho \Rrightarrow \tau)$ and strip$(\rho \Rrightarrow \tau) \subseteq \mathcal{C}(\rho \Rrightarrow \tau)$.

The rules generated by sorg$(\rho \geqslant \tau)$ are also *complete*: for every $\ell \Rrightarrow r \in \mathcal{R}(\rho \geqslant \tau)$ there is a $\ell' \Rrightarrow r' \in$ sorg$(\rho \geqslant \tau)$, a substitution γ and a context C such that $C[\ell'\gamma] = \ell$ and $C[r'\gamma] = r$. This directly follows from Lemmas 11 and 15.

3 Case Study: EVM bytecode

To demonstrate the applicability of our pipeline from Figure 1 we implement it in the context of Ethereum for EVM bytecode. We sketch how one could apply the approach to other smart contract languages in Section 5.

The EVM is a virtual machine formally defined in the Ethereum yellow paper [22]. It is based on a stack which holds bit vectors of size 256. The stack may over- or underflow;

both lead the EVM to enter an exceptional halting state. The EVM also features a volatile memory, which is a word-addressed byte array, and a persistent key-value storage, which is a word-addressed word array stored on the Ethereum blockchain.

3.1 Find Optimizations with ebso

We find optimizations using our tool ebso [19], an EVM bytecode superoptimizer. As an input ebso takes an ebso block – a basic block that additionally does not contain instructions whose semantics are not encoded, such as instructions that have an outside effect like `LOG`. Then, encoding the EVM execution state and unbounded superoptimization following Section 2.1, in the best case ebso produces a cheaper, observationally equivalent ebso block.

▶ **Example 16.** We encode the EVM execution state $\boldsymbol{\sigma}$ using four uninterpreted functions $\langle \mathsf{sk}, \mathsf{c}, \mathsf{hlt}, \mathsf{str} \rangle$ to model the stack, stack pointer, exceptional halting and storage:

(i) $\mathsf{sk}(j, \vec{x}, n)$ returns the word from position n, starting from 0, in the stack after executing j instructions on $\vec{x}$,

(ii) $\mathsf{c}(j)$ returns the number of words on the stack after executing j instructions,

(iii) $\mathsf{hlt}(j)$ returns true ($\top$) if exceptional halting has occurred after executing j instructions, and false ($\bot$) otherwise, and

(iv) $\mathsf{str}(j, \vec{x}, k)$ returns the word at key k after executing j instructions on $\vec{x}$.

Note that these functions represent all states throughout an execution, i.e., $\boldsymbol{\sigma}$, while to obtain σ_j for some j, we simply apply them to j thus: $\sigma_j = \langle \mathsf{sk}(j), \mathsf{c}(j), \mathsf{hlt}(j), \mathsf{str}(j) \rangle$. To refer to individual components of states we use subscripts, for instance we write sk_σ to refer to the stack of state σ.

For a program ρ which takes d arguments on the stack we add d fresh variables to represent the input $\vec{x}$ and add the following constraint to $\mathsf{enc\_init}(\vec{x}, \boldsymbol{\sigma})$:

$$\bigwedge_{0 \leqslant i < d} \mathsf{sk}_\sigma(\vec{x}, 0, i) = x_i \wedge \mathsf{c}_\sigma(0) = d \wedge \mathsf{hlt}_\sigma(0) = \bot$$

The storage str is initialized similarly using an Ackermann encoding [1, 14].

To ease readability and save space we do not include the EVM's memory in this encoding of the execution state. It can be represented analogously to the storage.

▶ **Example 17.** Next we instantiate the operational semantics of the instructions. The constraint $\mathsf{enc\_stack}(\iota, \sigma_j, \sigma_{j+1})$ describes the effect that ι has on stack. Here we give as example the instruction `SUB` and refer to [22] or [19] for details. Let $-_{\mathsf{bv}}$ denote subtraction on bit-vectors. Then we have

$$\begin{aligned} \mathsf{enc\_stack}(\texttt{SUB}, \sigma_j, \sigma_{j+1}) := {} & \mathsf{sk}_\sigma(j+1, \vec{x}, \mathsf{c}_\sigma(j+1) - 1) \\ & = \mathsf{sk}_\sigma(j, \vec{x}, \mathsf{c}_\sigma(j) - 1) -_{\mathsf{bv}} \mathsf{sk}_\sigma(j, \vec{x}, \mathsf{c}_\sigma(j) - 2) \end{aligned}$$

Using $\mathsf{enc\_stack}$ we can formulate the constraint $\mathsf{enc\_step}$. Here $\delta(\iota)$ and $\alpha(\iota)$ refer to the number of words which ι deletes from, and adds to the stack respectively. For all instructions except `SSTORE` we have:

$$\begin{aligned} \mathsf{enc\_step}(\iota, \sigma_j, \sigma_{j+1}) := {} & \mathsf{enc\_stack}(\iota, \sigma_j, \sigma_{j+1}) \wedge \\ & \mathsf{c}_\sigma(j+1) = \mathsf{c}_\sigma(j) + \alpha(\iota) - \delta(\iota) \wedge \\ & \forall n.\, n < \mathsf{c}_\sigma(j) - \delta(\iota) \rightarrow \mathsf{sk}_\sigma(j+1, \vec{x}, n) = \mathsf{sk}_\sigma(j, \vec{x}, n) \wedge \\ & \mathsf{hlt}_\sigma(j+1) = \mathsf{hlt}_\sigma(j) \vee \mathsf{c}_\sigma(j) - \delta(\iota) < 0 \vee \mathsf{c}_\sigma(j) - \delta(\iota) + \alpha(\iota) > 2^{10} \wedge \\ & \forall w.\, \mathsf{str}_\sigma(j+1, \vec{x}, w) = \mathsf{str}_\sigma(j, \vec{x}, w) \end{aligned}$$

Here the second line updates the counter for the number of words on the stack according to the number of words added and deleted. The third line expresses that all words on the stack below $\mathsf{c}_\sigma(j) - \delta(\iota)$ are preserved. The fourth line captures that exceptions relevant to the stack can occur through either an underflow or an overflow, and that once it has occurred, an exceptional halt state persists. Finally the last line states that all $\iota \neq$ `SSTORE` do not change the storage. The constraint for `SSTORE` is similar updating the storage using the Ackermann encoding.

▶ **Example 18.** The final ingredient we need to instantiate is the equivalence relation on states. For two states at steps j_1 and j_2 where $\sigma_{j_1} = \langle \mathsf{sk}(j_1), \mathsf{c}(j_1), \mathsf{hlt}(j_1), \mathsf{str}(j_1)\rangle$ and $\sigma'_{j_2} = \langle \mathsf{sk}'(j_2), \mathsf{c}'(j_2), \mathsf{hlt}'(j_2), \mathsf{str}'(j_2)\rangle$ and input $\vec{x}$ we define the constraint $\mathsf{enc\_eq\_state}(\sigma_{j_1}, \sigma'_{j_2})$ as

$$
\begin{aligned}
&\mathsf{c}(j_1) = \mathsf{c}'(j_2) \land \mathsf{hlt}(j_1) = \mathsf{hlt}'(j_2)\\
&\land \forall\, w.\, \mathsf{str}(j_1, \vec{x}, w) = \mathsf{str}'(j_2, \vec{x}, w)\\
&\land \forall\, n.\, n < \mathsf{c}(j_1) \to \mathsf{sk}(j_1, \vec{x}, n) = \mathsf{sk}'(j_2, \vec{x}, n)
\end{aligned}
$$

With the presented encoding, ebso, and an SMT solver we can now automatically find optimizations for EVM bytecode. Next, we also want to automatically generate rules.

3.2 Generate Rules with sorg

To generate rules for EVM bytecode we implemented sorg, a superoptimization based rule generator. Like ebso, sorg is implemented in OCaml; sorg depends on ebso for the representation of EVM bytecode and SMT encoding to check observational equivalence.

The main contribution of sorg is to provide notions of program schema, substitutions, and context in order to implement the two main procedures of Section 2.2: generalize and strip. For generalize we implement the procedure from Definition 10, keeping only the most general rules in the result.

▶ **Example 19.** In our evaluation in Section 4, we found the following optimization:

```
SWAP1 POP PUSH 0 PUSH 1 MUL PUSH 0 ⩾ SWAP1 POP PUSH 0 DUP1
```

Generalizing immediate arguments and dropping the prefix `SWAP1 POP` sorg yields two optimization rules: `PUSH` x `PUSH 1 MUL PUSH` $x \Rrightarrow$ `PUSH` x `DUP1` as well as `PUSH 0 PUSH` x `MUL PUSH 0` $\Rrightarrow$ `PUSH 0 DUP1`.

For strip we implement the procedure from Definition 14, keeping only the most stripped rules.

▶ **Example 20.** From the rule `CALLVALUE DUP1 POP` $\Rrightarrow$ `CALLVALUE CALLVALUE POP` sorg can either strip the postfix `POP` or the prefix `CALLVALUE`, obtaining the rules `CALLVALUE DUP1` $\Rrightarrow$ `CALLVALUE CALLVALUE` and `DUP1 POP` $\Rrightarrow$ `CALLVALUE POP`.

One main ingredient of both generalize and strip is a check for observational equivalence. To determine observational equivalence in sorg we use an SMT encoding with components from ebso, similar to Definition 7. For two program schemas ρ and τ, we have $\rho \equiv \tau$ if there are no inputs that distinguish them. That is

$$
\begin{aligned}
&\exists \vec{x}.\, \mathsf{enc\_init}(\vec{x}, \boldsymbol{\sigma}) \land \mathsf{enc\_init}(\vec{x}, \boldsymbol{\sigma'})\\
&\qquad \land \mathsf{enc\_progr}(\rho, \boldsymbol{\sigma}) \land \mathsf{enc\_progr}(\tau, \boldsymbol{\sigma'})\\
&\qquad \land \neg \mathsf{enc\_eq\_state}(\sigma_{|\rho|+1}, \sigma'_{|\tau|+1})
\end{aligned}
$$

With sorg we can now automatically generate rules, but it remains to glue the tools together and implement a feedback mechanism.

3.3 Coordinate with ppltr

To coordinate our tools ebso and sorg we implemented the tool ppltr, a populator for a peephole optimizer. As ebso and sorg, ppltr is implemented in OCaml. The tool has two main tasks. The first is to manage the interfaces, i.e., to generate ebso blocks from smart contracts, generate ebso blocks for a given size k, prepare optimizations generated by ebso as input for sorg, and analyze and de-duplicate a set of rules produced by sorg. The second main task is to feed back the optimization rules, i.e., to rewrite right-hand sides of the optimization rules themselves, and apply the optimization rules to ebso blocks. To achieve the latter task, ppltr implements a rewrite engine.

4 Evaluation

We evaluate our pipeline by generating optimization rules for EVM bytecode. We collected the 250 most called smart contracts until block 9 786 000 at Apr-01-2020 12:17:26 PM +UTC from the Ethereum blockchain using Google BigQuery.[3]

We split the 250 contracts into 106 798 ebso blocks $\mathcal{E}$. As peephole optimization rules typically span only few instructions, we restrict the size of a block: using a sliding window we split every block larger than 6 instructions into k blocks of at most 6 instructions. To reduce the noise, we remove blocks which are only different in the arguments of PUSH keeping only those with words of size smaller than 5 bit. We so obtain 54 301 ebso blocks.

(1) Using ebso find 1580 optimizations from these blocks, run on a cluster with Intel Xeon Gold 6126 CPUs at 2.60 GHz, 2 GB of memory and a time-out of 15 min.

(2) From these optimizations, we generate 1525 rules with sorg, run on the same set-up. For 48 optimizations sorg timed out and could not generate rules and we removed roughly half the rules, as they were duplicates generated from different optimizations.

(3) Thus we arrive at 758 rules $\mathcal{R}_0$, which we use with the rewrite engine of ppltr to

  (a) rewrite the right-hand sides of $\mathcal{R}_0$ reducing 4 rules, and

  (b) rewrite our original ebso blocks in $\mathcal{E}$, which changed 17 255 ebso blocks.

We again use the same window-size and noise reduction to get 25 585 new ebso blocks. Going through the same procedure, we find 452 optimizations with ebso, and generate 435 rules $\mathcal{R}_1$ with sorg with 16 timeouts. Combining the results we get 993 rules $\mathcal{R}_2 = \mathcal{R}_0 \cup \mathcal{R}_1$ which are available at

`github.com/mariaschett/ppltr/blob/v1.0/eval/17-reduced-rules.csv`

We right-reduced 31 rules in $\mathcal{R}_2$ and discarded 967 replicated rules originating from different optimizations. One optimization generated two rules (*cf.* Example 19).

To estimate gas and size saving on a contract level we apply the rules in $\mathcal{R}_2$ to

1. our original 250 most called smart contracts, and
2. extend the data set to the 1000 most called contracts.

[3] `cloud.google.com/blog/products/data-analytics/ethereum-bigquery-public-dataset-smart-contract-analytics`.

Table 1 Savings when applying the rules in $\mathcal{R}_2$ on most called contracts.

	accumulated gas savings	accumulated length savings	
250 most called contracts	106 811 g	35 699 instructions	3.94 %
1000 most called contracts	435 002 g	146 376 instructions	4.58 %

Table 1 shows our results. The first column shows the accumulated gas savings over all contracts, and the second column shows the accumulated length savings. Note that results depend on the order in which the rules are applied (*cf.* Section 5). First, we can observe that the rules translate well from 250 to 1000 contracts, achieving roughly 4 times higher savings, which demonstrates that $\mathcal{R}_2$ also extends beyond the original data set, from which it was generated.

Now let us consider the gas savings. In Table 1 we accumulate the cost of all the removed instructions for each contract. How much is actually saved, however, depends on how often the contract is called and which parts are executed. Unfortunately we lack the resources to replay all the transactions to determine the exact savings. Taking into account how often a contract was called, we save 7.41×10^{10} g for the former and 1.02×10^{11} g for the latter. Assuming that about 10 % of a contract is executed per call and that savings are uniformly distributed, this translates to 41 049.33 $ and 56 505.15 $ for a gas price of 27.6 gwei and an ETH-USD course of 200.62 $, which are averages from `etherscan.io/charts`.

While the cost of executing a cheap instruction like `ADD` or `POP` may be negligible, the cost of storing that instruction may not be so. Therefore, we also look at the savings in length: the overall storage space of the bytecode reduces by more than 4.5 %. The contract with the highest length saving was reduced by 19.94 %, removing 345 from originally 1730 instructions.

We also analyze which rules are applied to the contracts. Applying rules may lead to the applicability of other rules, but exploring all rewrite sequences is intractable, and we assume that initial applicability on a contract is a reasonable proxy. Figure 2 groups rules in $\mathcal{R}_2$ by their applicability to the 1000 most called contracts. We can observe a long tail: more than half of the nearly $1k$ rules are applicable only 10 times or less, whereas the top 50 rules are applicable more than 500 times. This suggests that, if a smaller set of rules is desired, this analysis can guide which rules to discard.

Next we inspect the rules within $\mathcal{R}_2$. The five most applied rules for the 1000 most called contracts are listed in Figure 3. Most of these rules are relatively simple and should clearly be applied exhaustively. The fourth rule is perhaps a bit unexpected and may have been missed on manual inspection, but it is cheaper to execute `CALLVALUE` twice than duplicating its result. The last rule hints at a specific compiler produced anti-pattern. Our approach could also be leveraged to detect those.

Figure 4 shows the six rules with the highest gas savings, 17 g and 15 g. We consider two of these rules in more detail. The rule `PUSH 1 MUL PUSH 0 NOT AND` $\Rightarrow \varepsilon$ combines two observations – that 1 and `PUSH 0 NOT` are neutral elements for multiplication and `AND` respectively.

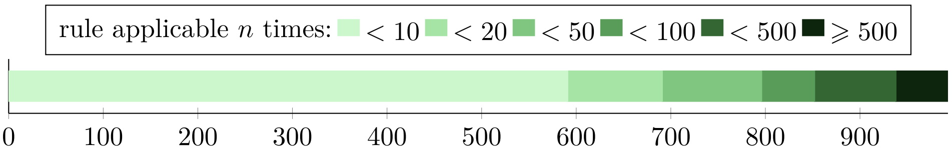

Figure 2 Applicability of rules in $\mathcal{R}_2$ to 1000 most called contracts.

1. SWAP1 POP POP $\Rrightarrow$ POP POP (×8926)
2. ISZERO ISZERO ISZERO $\Rrightarrow$ ISZERO (×7893)
3. PUSH y PUSH x SWAP1 $\Rrightarrow$ PUSH x PUSH y (×7742)
4. CALLVALUE DUP1 $\Rrightarrow$ CALLVALUE CALLVALUE (×7740)
5. SWAP1 SLOAD SWAP1 PUSH x EXP SWAP1 $\Rrightarrow$ PUSH x EXP SWAP1 SLOAD (×5625)

Figure 3 Rules most applied to the 1000 most called contracts.

1. PUSH 1 MUL DUP3 PUSH 0 NOT AND $\Rrightarrow$ DUP3
   PUSH 1 MUL PUSH 0 NOT AND $\Rrightarrow$ ε
2. PUSH 0 DUP6 DUP5 SUB LT ISZERO $\Rrightarrow$ PUSH 1
   PUSH 0 NOT AND EQ ISZERO ISZERO $\Rrightarrow$ EQ
   SWAP1 PUSH 0 NOT AND SWAP1 $\Rrightarrow$ ε
   PUSH 0 DUP2 PUSH x AND LT ISZERO $\Rrightarrow$ PUSH 1

Figure 4 Rules saving most gas.

Depending on the implementation of the peephole optimizer it may be desirable to split this rule which could be achieved by left-reducing the rules. Key to the rule PUSH 0 DUP6 DUP5 SUB LT ISZERO $\Rrightarrow$ PUSH 1 is the less-than comparison LT with the smallest element 0 always evaluating to false. The rule does not depend on the result of DUP6 DUP5 SUB, and indeed this is replaced by DUP2 PUSH x AND in the otherwise identical rule in the last line. Generalizing those two rules would require the use of higher-order patterns.

Rules may not only save gas, but also reduce the length of the produced code. These often coincide, and indeed the top 14 length-reducing rules, removing 5 instructions each, subsume the above gas-saving rules. On the other end, there are also rules which save gas but do not reduce the length such as CALLVALUE DUP1 $\Rrightarrow$ CALLVALUE CALLVALUE saving 1 g. In Table 2, we analyze the right-hand sides of $\mathcal{R}_2$. We investigated which instructions were *added*, i.e., do not appear on the left-hand side, and *removed*, i.e., appear on the left- but not the right-hand side of the rule. We group instructions for arithmetic, comparison, bitwise operations, and environment/memory. Unsurprisingly, many more instructions were removed than added, which is expected, because removing instructions always saves gas. The majority of removed instructions is concerned with the stack layout. Surprisingly, also ISZERO is often redundant – as also observed in the second rule in Figure 3. Still, instructions are also synthesized on the right-hand side giving rise to optimizations taking the semantic of an instructions into account – potentially also interacting with stack manipulation, for example the rule SWAP1 LT $\Rrightarrow$ GT.

Finally, we also successfully validated all rules $\mathcal{R}_2$ by running a reference implementation of the EVM, go-ethereum version 1.9.14 on pseudo-random input.[4] Therefore, we run the bytecode of every block in $\mathcal{E}$ and the bytecode obtained by applying the rewrite rules to observe that both produce the same final state.

5 Related and Future Work

Chen et al. [7] also developed a tool to rewrite optimization patterns in EVM bytecode. As opposed to our approach, they devised their 24 (anti-)patterns by manual inspection of the code base. Albert et al. [2] synthesize optimized straight-line EVM bytecode for operations on

[4] github.com/ethereum/go-ethereum

Table 2 Added and removed instructions by group.

	arith.	comp.	`ISZERO`	bitwise	`DUP`*i*	`SWAP`*i*	`PUSH`	`POP`	env./mem.
added	10	27	24	12	47	28	134	14	29
removed	80	92	108	83	345	952	182	173	18

the stack with Max-SMT. To gain efficiency, they do not encode the semantics of bit-vector instructions, and instead employ hand-crafted simplification rules. These hand-crafted rules could be inspired by, or even automatically derived from, rules generated by ppltr, which do consider the semantic of bit-vector instructions. Bansal et al. [5] use superoptimization to automatically generate a peephole optimizer for x86 binaries. Aside from the application, the main difference of their approach to ppltr is that it does not process optimizations into rules but instead keeps them in an optimization database in order to reapply them. Moreover it uses an enumeration based superoptimizer, which is more exhaustive, but limits instruction sequences to length 3.

We believe our approach is also applicable for different smart contract languages. Facebook's Move [6] is a gas-metered and verification friendly designed language with an existing code base, such as for example from `github.com/libra/libra/tree/master/language/move-lang/functional-tests/tests`. The machine model of Move is stack-based with typed locals. To adapt the presented approach the SMT encoding would need to be extended to incorporate types and locals. Michelson [15], the smart contract language for the Tezos blockchain, also comes with a detailed formal semantics. Like the EVM it is a stack-based language, but features high-level data types, like lists, sets, and maps. To use the presented approach these data types need to be handled in the SMT encoding and SMT solvers do support complex theories such as sets and lists. Moreover, type information could be used to prune search space, resulting in a positive performance impact.

To automatically integrate the rules generated by ppltr into a compiler a DSL like the one used by GCC[5] or Alive [16] might prove useful. Such an automatic integration would be especially welcome when one wants to re-populate the optimizer of a compiler, e.g. because new instructions are available, such as the addition of shift-operators to the EVM.

Hirai [10] used the meta-tool Lem [17] to formalize the semantics of the EVM. This formalization was extended by Amani et al. [3] by a program logic using the interactive proof assistant Isabelle/HOL to provide an approach to the verification of Ethereum smart contracts. Another formalization of the EVM semantics by Hildenbrandt et al. [9] use the K-framework [20], a rewriting-based framework for defining programming language design and semantics. One of these formalizations could be used to verify the correctness of our encoding, or possibly even generate it automatically.

Our definitions in Section 2.2 are based on concepts from term rewriting [4] and thus we also look at the machinery of term rewriting. Termination of the rules ensures we can apply them exhaustively without looping. Intuitively all rules in $\mathcal{R}_2$ are terminating, since left-hand sides have a higher cost than right-hand sides, and indeed the termination prover WANDA [12] shows termination of all 993 rules in $\mathcal{R}_2$.[6] Confluence guarantees a unique result regardless of how the rules are applied. To check confluence one analyses critical pairs, situations where

[5] `gcc.gnu.org/onlinedocs/gccint/The-Language.html`

[6] We chose WANDA as its support for types allowed us to leverage that arguments of `PUSH` are words, which greatly aided the automated proof.

application of one rule potentially destroys the possibility for applying another one. The confluence checker CSI [18] reports 82 765 critical pairs, 14 973 of which are joinable and thus harmless. The remaining 67 792 are not, so the rules in $\mathcal{R}_2$ are not confluent. This is not surprising, since there are different ways to achieve the same with the same cost, e.g. `PUSH` x `PUSH` x and `PUSH` x `DUP1`. This may be resolved by defining an additional precedence on the rules, e.g., based on the size of their bytecode. To make a terminating set of rules confluent, one can use *completion* – automatically if we employing tools such as Ctrl [21]. Finally, one could imagine more expressive rules such as `PUSH` x `PUSH` y `ADD` $\Rrightarrow$ `PUSH` z *where* $z = x + y$. Such rules allow to capture constant folding. To do so, rules in *constrained rewriting* [13] come with constraints over a theory as used in SMT solvers.

6 Conclusion

We propose a pipeline to populate the peephole optimizer of a smart contract compiler with three phases to

(1) find optimizations, from which we
(2) generate rules, and
(3) a feedback mechanism to apply the rules.

We demonstrate our approach for EVM bytecode using the tools ebso, sorg, and ppltr, generating 993 peephole optimization rules from the 250 most called contracts of the Ethereum blockchain. We successfully applied our rules to the 1000 most called contracts and discarded 146 376 instructions, saving 435 002 g and 4.5 % storage space. An advantage of our approach lies in its modularity. On the one hand in the modularity of the phases. One could, for example, obtain additional optimizations in a different manner and incorporate them easily. On the other hand, there is the modularity inherent to peephole optimization rules being applied to short programs: it enables an iterative approach to encoding and optimizing instructions based on feasibility and profitability.

Our approach is tailored towards new, rapidly evolving languages and their compilers with clear cost models such as gas metering, and we believe readily applies to languages other than EVM bytecode such as Move and Michelson.

References

1 Wilhelm Ackermann. *Solvable cases of the decision problem.* Studies in logic and the foundations of mathematics. North-Holland Publishing Co., Amsterdam, 1954.

2 Elvira Albert, Pablo Gordillo, Albert Rubio, and Maria A Schett. Synthesis of super-optimized smart contracts using Max-SMT. In *Proc. 32nd CAV*, volume 12224 of *LNCS*. Springer, 2020. doi:10.1007/978-3-030-53288-8_10.

3 Sidney Amani, Myriam Bégel, Maksym Bortin, and Mark Staples. Towards verifying ethereum smart contract bytecode in Isabelle/HOL. In *Proc. 7th CPP*, pages 66–77. ACM, 2018. doi:10.1145/3167084.

4 Franz Baader and Tobias Nipkow. *Term Rewriting and All That.* Cambridge University Press, New York, NY, USA, 1998.

5 Sorav Bansal and Alex Aiken. Automatic generation of peephole superoptimizers. In *Proc. 12th ASPLOS*, pages 394–403. ACM, 2006. doi:10.1145/1168857.1168906.

6 Sam Blackshear, Evan Cheng, David L Dill, Victor Gao, Ben Maurer, Todd Nowacki, Alistair Pott, Shaz Qadeer, Dario Russi, Stephane Sezer, Tim Zakian, and Runtian Zhou. Move: A language with programmable resources. developers.libra.org/docs/assets/papers/libra-move-a-language-with-programmable-resources/2020-04-09.pdf.

7 Ting Chen, Zihao Li, Hao Zhou, Jiachi Chen, Xiapu Luo, Xiaoqi Li, and Xiaosong Zhang. Towards saving money in using smart contracts. In *Proc. 40th ICSE-NIER*, pages 81–84. ACM, 2018. doi:10.1145/3183399.3183420.

8 Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proc. 14th TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008. doi:10.1007/978-3-540-78800-3_24.

9 Everett Hildenbrandt, Manasvi Saxena, Xiaoran Zhu, Nishant Rodrigues, Philip Daian, Dwight Guth, Brandon Moore, Yi Zhang, Daejun Park, Andrei Ştefănescu, and Grigore Roşu. KEVM: A complete semantics of the ethereum virtual machine. In *Proc. 31st CSF*, pages 204–217. IEEE, 2018. doi:10.1109/CSF.2018.00022.

10 Yoichi Hirai. Defining the Ethereum virtual machine for interactive theorem provers. In *Financial Cryptography and Data Security*, volume 10323 of *LNCS*, pages 520–535. Springer, 2017. doi:10.1007/978-3-319-70278-0_33.

11 Abhinav Jangda and Greta Yorsh. Unbounded superoptimization. In *Proc. Onward! 2017*, pages 78–88. ACM, 2017. doi:10.1145/3133850.3133856.

12 Cynthia Kop. *Higher Order Termination*. PhD thesis, Vrije Universiteit, Amsterdam, 2012.

13 Cynthia Kop and Naoki Nishida. Constrained term rewriting tool. In *Proc. 20th LPAR*, volume 9450 of *LNCS*, pages 549–557, 2015. doi:10.1007/978-3-662-48899-7.

14 Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer, Berlin, 2008. doi:10.1007/978-3-540-74105-3.

15 Nomadic Labs. Michelson: the language of smart contracts in Tezos. tezos.gitlab.io/whitedoc/michelson.html.

16 Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. Practical verification of peephole optimizations with Alive. *Commun. ACM*, 61(2):84–91, 2018. doi:10.1145/3166064.

17 Dominic P. Mulligan, Scott Owens, Kathryn E. Gray, Tom Ridge, and Peter Sewell. Lem: Reusable engineering of real-world semantics. In *Proc. 19th ICFP*, pages 175–188. ACM, 2014. doi:10.1145/2628136.2628143.

18 Julian Nagele, Bertram Felgenhauer, and Aart Middeldorp. CSI: New evidence – A progress report. In *Proc. CADE-26*, volume 10395 of *LNAI*, pages 385–397, 2017. doi:10.1007/978-3-319-63046-5.

19 Julian Nagele and Maria A. Schett. Blockchain superoptimizer. In *Preproc. 29th LOPSTR*, pages 166–180, 2019. arXiv:2005.05912.

20 Grigore Roşu and Traian Florin Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010. doi:10.1016/j.jlap.2010.03.012.

21 Sarah Winkler and Aart Middeldorp. Completion for logically constrained rewriting. In *Proc. 3rd FSCD*, volume 108 of *LIPIcs*, pages 30:1–30:18, 2018. doi:10.4230/LIPIcs.FSCD.2018.30.

22 Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger, 2018. Byzantium Version e94ebda, ethereum.github.io/yellowpaper/paper.pdf.