Optimal Synthesis of Michelson Bytecode

Tianchi Yu
Institut Polytechnique de Paris, Palaiseau, France
Nomadic Labs, Paris, France
https://www.ip-paris.fr/
https://www.nomadic-labs.com/

Abstract

A Smart Contract is a program that is executed by every node participating in a blockchain. To account for the computational cost of this execution, a smart contract consume gas, an abtract resource purchased by the users of the blockchain. They consume a lot of gas, an abstract resource purchased through cryptocurrency. There is therefore economic incentives to reduce gas consumption. Michelson is a stack-based, strictly typed language in which Smart Contracts of the Tezos blockchain are written to ensure the safety of the Tezos blockchain. This report implements a blackbox optimizer for Michelson programs based on S-metaheuristics.

Author: Please fill in 1 or more \ccsdesc macro

Keywords and phrases Smart Contract, Michelson program, Optimization, S-metaheuristics, Translation Validation

Acknowledgements I would like to thank the entire Nomadic Labs team for their welcome. I am delighted that I have the opportunity to have this great experience in one extremely enriching research environment. My gratitude goes in particular to my supervisors, Richard Bonichon and Yann Régis-Gianas, for their pedagogy, and their clarity, which allowed me to learn a lot.

1 Introduction

1.1 Smart Contracts

1.1.1 What is a Blockchain?

A blockchain is a type of database. It differs from a typical database in the way it stores information. Blockchains store data in blocks that are then chained together. As new data comes in it is entered into a fresh block. Once the block is filled with data it is chained onto the previous block, which makes the data chained together in chronological order. Arranging transactions in chronological order prevents double-spending, which is required by financial accounting.

Cryptocurrencies of all types make use of distributed ledger technology known as blockchain. Blockchains act as decentralized systems for recording and documenting transactions that take place involving a particular digital currency. Put simply, a blockchain is essentially a digital ledger of transactions that is duplicated and distributed across the entire network of computer systems on the blockchain. Each block in the chain contains a number of transactions, and every time a new transaction occurs on the blockchain, a record of that transaction is added to every participant's ledger. The decentralised database managed by multiple participants is known as Distributed Ledger Technology (DLT).

Tezos

Tezos [3] is a decentralized, open-source *Proof of Stake* (see Note 1) blockchain network and it supports Smart Contracts and *tez* crypto-currency (XTZ). Its characteristic is to natively support protocol updates without hard forks. The Tezos blockchain environment is based on

OCaml. All the programs in this report are also implemented in OCaml with the help of some tools in the Tezos codebase.

▶ Note 1. Proof of Stake (PoS) protocols are a class of consensus mechanisms for blockchains that work by selecting validators in proportion to their quantity of holdings in the associated cryptocurrency. Unlike a Proof of Work (PoW) protocol, PoS systems do not incentivize extreme amounts of energy consumption.

1.1.2 What is a Smart Contract?

A Smart Contract is a computer agreement or program designed to spread, verify or execute contracts in an information-based way. Smart contracts in blockchain have the following characteristics: Rules are transparent, and data in the contract are visible to the outside; All transactions are publicly visible, and there will be no false or hidden transactions, thus cannot be modified.

Smart Contracts are often regarded as a powerful application of blockchain technology. These contracts are actually computer programs that can monitor all aspects of the agreement. When the conditions are met, the Smart Contract can be fully self-executing and self-enforcing. These tools provide safer and more automated alternatives than traditional contract law, as well as faster and cheaper applications than traditional methods.

Michelson

The Tezos blockchain has a rather low-level bytecode Smart Contract language called Michelson [2]. Michelson is a domain-specific language that is both stack based and strongly typed. This specification gives a detailed formal semantics of the Michelson language and a short explanation of how Smart Contracts are executed and interact in the blockchain.

1.1.3 What is the Purpose of *Gas*?

On the Tezos network, Michelson programs consume *gas*, which is an abstract resource designed to bound Smart Contract computation time and thus (amongst other things) incentivize efficient use of on-chain computation. Specifically, *gas* represents computational cost related to a transaction, an amount of *gas* is assigned to different instructions. The main goal of *gas* is to be a security measure against DoS (i.e. Denial-of-Service attack), because an unbounded execution would block nodes and wouldn't allow the chain to move forward, so it offers liveness guarantee for blockchain network.

1.2 Optimizing Gas Consumption of Smart Contracts

The objective of this internship is to optimize Michelson Bytecode with respect to gas consumed. Specifically, we study super-optimization [6, 14] (finding global program optimizations which might be missed by a smaller and simpler search for local optimizations). We aim for an heuristics-based method using S-metaheuristics [15] to find the optimal bytecode in a fully blackbox way.

1.2.1 Overview

A Michelson program (e.g. Listing 1) can be seen as a series of instructions that are run in sequence, each instruction receives as input the stack resulting from the previous instruction, and rewrites it for the next one. Every Michelson program has two arguments, `parameter`

Listing 1 One Michelson Program

```
parameter int;
storage int;
code { DROP ; PUSH int 47; NEG ; PUSH int 84 ;
       SWAP; SWAP ; DROP ; NIL operation ; PAIR }
```

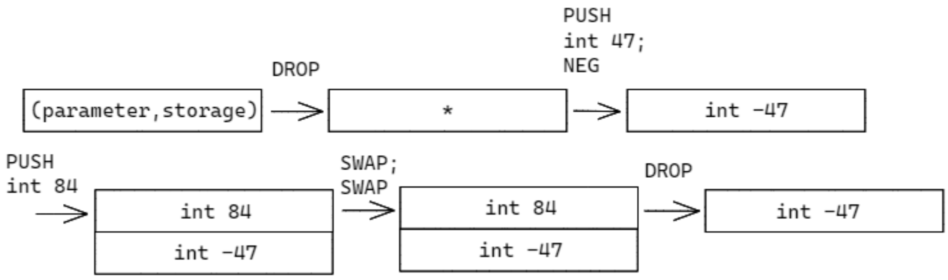

Figure 1 Evolution of the stack

and `storage`, represented as a pair of values on the top of the stack before execution of the program.

We present some basic examples about how Michelson programs execute. Instructions refer to Michelson primitives such as `DROP`, it means that drop the top element of the stack. `PUSH` instruction pushes a constant value of a given type onto the stack; `NEG` intends to negate a numerical value ; `SWAP` effects two nearby elements on the stack and swap their positions; `NIL operation; PAIR` is usually used at the end of one Michelson program, `NIL` is an opcode that adds an empty list of the specified type (e.g. `operation`) on top of the stack, and *PAIR* takes the two elements on top of the stack, creates a new pair containing these two elements, and pushes back the pair on the stack.

This basic overview for Michelson language helps us understand that the program in the Listing 1 can be optimized. For example, `SWAP; SWAP` swaps two values in the stack twice, which means there is no change. There are values in the stack directly dropped by `DROP` instruction. Therefore, the effect of original program should be equivalent to directly `PUSH int -47`, which is the optimal program we expect to have. Fig.1 shows the evolution of the stack.

Super-optimization is an idea to produce perfectly optimal code, in place of the code we currently have. It is typically done via a brute-force search of every possible instruction sequence, checking whether it performs the desired actions if it is the optimal one. This is costly, and thus impractical for general-purpose compilers. Thus we aim to explore the huge search space by building an heuristics-based method with S-metaheuristics also known as *Single-solution based metaheuristic algorithm.*

S-metaheuristics

When only one solution is being developed mathematically, and transformed by way of various stochastic or deterministic processes, the process is classified as an S-metaheuristic [7, 15]. S-metaheuristics can be advantageously used to solve such optimization problems. A wide range of heuristics exists (Hill Climbing, Random Walk, Metropolis Hasting [13] and Simulated Annealing, etc.). They iteratively improve a candidate solution by testing its "neighbors" and moving along the search space. Because solution improvement is evaluated

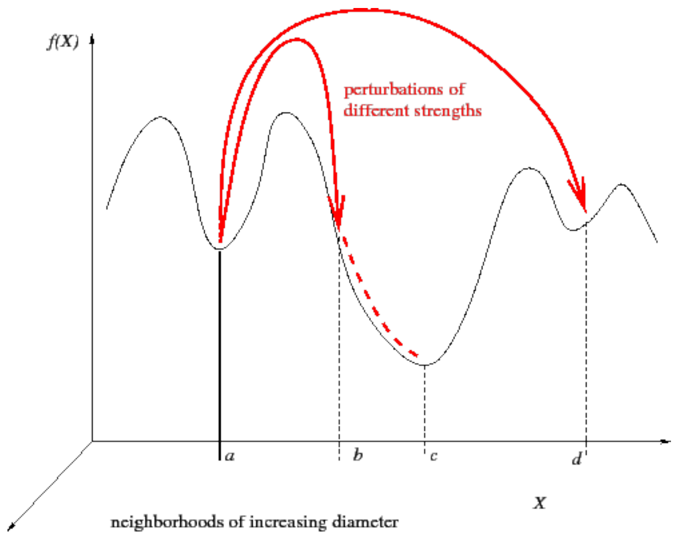

Figure 2 Iterated Local Search

by the objective function, it is said to guide the search.

Iterated Local Search (ILS)

Iterated Local Search [10] is based on building a series of local optimal solutions by disturbing the current local minimum and applying local search after starting from the modified solution.

Some S-metaheuristics are likely to fall into local optima, so the result depends on the initial input selected. Iterative local search fixes this issue by looking for iterations and the ability to restart from the best solution seen before. Note that ILS is configured by another search heuristic (for us: Hill Climbing). Once the local optimal value is found through this edge search, ILS will disrupt it and use the perturbed solution as the initial state of the edge search. At each iteration, ILS also records the best solution found. Unlike most other S-metaheuristic, if the research follows a misleading path, ILS can restore the best solution yet to start over from a healthy state.

1.2.2 Why Optimizing Contract is Important?

Smart Contracts that execute on the blockchain are critical. As we have discussed, *gas* costs by Smart Contracts are meant to equate to computation, e.g. if one instruction takes twice as much computation time/resources, it should consume twice as much *gas*, hence reducing *gas* consumption allows reducing paid fees. Thus developers must pay meticulous attention to the gas spent by their Smart Contracts, we thus need optimization tools that must be capable of effectively reducing the gas consumed by the Smart Contracts.

1.2.3 State of the Art

There are currently some research work on the super-optimization of Smart Contracts and most of them work on the Ethereum blockchain. E. Albert et al. [5] present an approach for super-optimization of Smart Contracts based on Max-SMT(Current Maximum Satisfiability [16]) which has two main phases : extraction of a stack functional specification from the basic blocks of the Smart Contract and then synthesis of optimized blocks by means of an efficient

Max-SMT encoding. J. Nagele and M.A. Schett [11] superoptimize EVM (i.e. Ethereum Virtual Machine) bytecode by encoding the operational semantics of EVM instructions as SMT formulas and leveraging a constraint solver to automatically find cheaper bytecode.

Considering only super-optimization, Eric Schkufza et al. [14] of Stanford University formulate the loop-free binary super-optimization task as a stochastic search problem. They encode competing constraints of transformation correctness and performance improvement to cost function, then use a Markov Chain Monte Carlo sampler to explore the space of all possible programs to find one that is an optimization of a given target program.

1.3 This Internship

1.3.1 Nomadic Labs

I was able to complete this internship in the team of Nomadic Labs, a research and development company, which contributes in particular to the implementation of the software core of the Tezos blockchain, and to the development of the language of the associated smart-contracts, Michelson.

1.3.2 Contributions

The approach presented by this report is basically split into three phrases: (i) sampling, (ii) search, (iii) proof.

Sampling

To apply S-meraheuristics method, we need a cost function that aims to guide the search. To establish this cost function, we choose to take inputs-outputs relationships as arguments of it. Generation of inputs-outputs pairs is realized by a Monte Carlo-based sampler and a Michelson interpreter. The use of a sampler is needed, because manually defining the inputs for each contract is at best impractical. There is an existing value sampler in Tezos codebase and we adapt this sampler to generate the corresponding input value for each contract randomly.

Search

The search starts from the empty program of Michelson language. Each program synthesized is scored by its distance of outputs with the expected one. Lower distance means higher score and a distance of zero is highly expected to obtain.

In my internship, ILS algorithm is implemented for this search process. The best programs found by Local Search are perturbed to more possible programs and applied Iterated Local Search. All programs with zero distance and less gas consumed are considered as candidates waiting for the proof of semantic equivalence with the original program.

Proof

Candidates found by the last step cannot be taken as correct solutions (i.e. optimized programs), because it is clear that having not enough inputs-outputs pairs can sometimes generate a program that is not equivalent, hence we implement Translation Validation (as defined below) to prove semantic equivalence between the source program and the candidate optimized program.

▶ **Definition 2.** *Translation Validation [12] is a technique for ensuring that the target code produced by a translator is a correct alternative representation of the same computation. Rather than verifying the translator itself, Translation Validation validates the correctness of each translation, generating a formal proof that it is indeed a correct [9].*

Translation Validation is proved by Z3 [4] SMT Solver in this report. Z3 is an efficient SMT Solver freely available from Microsoft Research. It is usually used in various software verification and analysis applications. Working as an SMT Solver, it is able to decide the satisfiability of formulas in a variety of theeories. We choose this tool to achieve our requirements. With its OCaml API, we can apply it in Tezos ecosystem.

▶ Note 3. Satisfiability modulo theories (SMT) generalizes boolean satisfiability (SAT) by adding equality reasoning, arithmetic, fixed-size bit-vectors, arrays, quantifiers, and other useful first-order theories [8].

For the whole tool built at the end, the input and output should be Michelson programs, the only difference is that the output program consumes less gas. The execution results show in the Section 5, where we find several optimal programs for different original programs. On the other hand, the tool is still limited by its efficiency. For some complex Smart Contracts, the optimization process may take a lot of time, which should be able to optimize by a better implementation of S-metaheuristics.

2 Sampler : Generation of Inputs/Outputs Relationships

In this section, we parse Michelson programs and interpret them to obtain a sets of inputs-outputs pairs. It can help synthesize optimized Michelson contracts. The program synthesized by the search algorithm takes the generated input values as inputs, and it will be considered as a candidate only if the output values are consistent with the expected output values.

2.1 Parse Michelson Programs

The concrete syntax of Michelson is called as Micheline. Thus the abstract syntax tree of the Michelson program is constructed by `('l,'p) node` in Micheline, where `'l` stands for location and `'p` stands for primitives of node. The definition of this type is in Listing 2. And Listing 3 shows an example of structure of a AST for a Michelson program. In this structure, `K_parameter` and `K_storage` are primitives separately for two arguments of Michelson programs. And `T_int` is a primitive for type of integer.

Micheline

Micheline [1] is a data format comparable to JSON, XML, S-expressions, and YAML. Its main purpose is to serve as the concrete syntax for the Michelson language. The structure of a Micheline node is simple, it is a node can only be one of the five following constructs: An integer in decimal notation; A character string delimited by the double quotation character "; A byte sequence in hexadecimal notation prefixed by 0x; The application of a primitive to a whitespace-delimited list of nodes and annotations.; A sequence of nodes delimited by curly braces (and) and separated by semi-colons (;).

2.2 Sampler Generation and Boundaries

Using the script translator tool in Tezos codebase, we parse Michelson program to get types of *parameter* and *storage* arguments. Then these types of values are able to be generated by

Listing 2 type of ('l, 'p) node

```
type annot = string list

type ('l, 'p) node =
  | Int of 'l * Z.t
  | String of 'l * string
  | Bytes of 'l * Bytes.t
  | Prim of 'l * 'p * ('l, 'p) node list * annot
  | Seq of 'l * ('l, 'p) node list
```

Listing 3 AST of Michelson program

```
Seq
  (0,
    [ Prim (1, K_parameter, [Prim (2, T_int, [], [])], []);
      Prim (3, K_storage, [Prim (4, T_int, [], [])], []);
      Prim
        ( 5,
          K_code,
          [ Seq
            ( 6,
              [ Prim (7, I_NIL, [Prim (8, T_operation, [], [])], []);
                Prim (9, I_PAIR, [], []) ] ) ],
          [] ) ] )
```

`Michelson_value_sampler` Module (see Appendix A.1). The size of values can be limited by bounds inside of the module of parameters.

▶ Remark 4. The Michelson Sampler is able to generate a variety of types of value. But in this work, integers are considered as the most important types, as they are easy to be manipulated and observed, and also arithmetic operations of integers are very important part for transaction in Smart Contracts.

2.3 Interpreter and *Gas* Consumed

Once the input values are generated, we interpret Michelson program to get the output. We pair the input and output values one by one, combine them into a set of input and output relationships and store them in a json file.

In addition, the interpreter (see Appendix A.2) also provides a function calculates the gas consumption of each Michelson program. This function is key to evaluate whether we indeed have optimized our original program. To compute the gas consumed, an initial global gas is set. We are able to obtain the remaining gas after each execution of programs. A simple subtraction returns what we want.

▶ Remark 5. The Michelson program fed to the interpreter must be well-typed, otherwise it cannot be executed. Therefore, in the process of rewriting and verifying the program later, whether a Michelson program is well-typed or ill-typed needs to be discussed.

2.4 Implementation and Examples

The main tools used are Michelson parser, Michelson interpreter and Michelson Sampler. These three tools are respectively constructed by three modules of `parse_parameters_storage`,

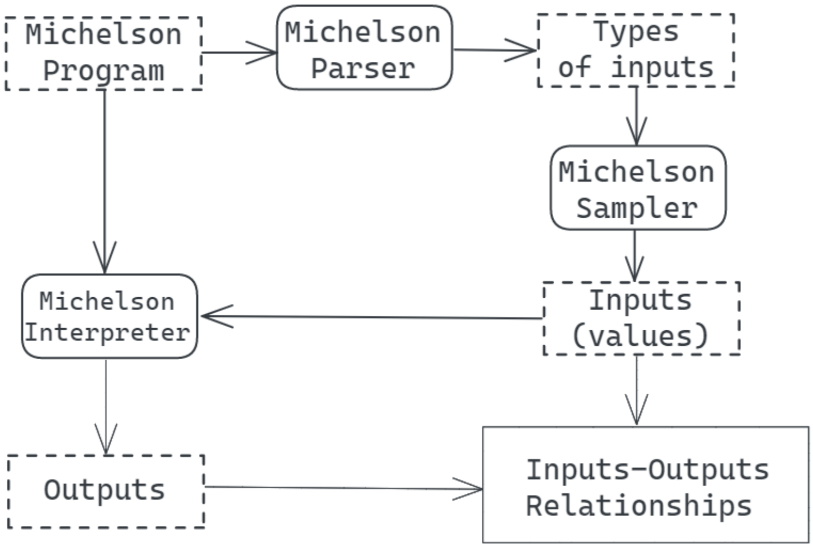

Figure 3 Samping Process

`michelson_value_sampler`, `michelson_interpreteur` in my code [17]. Figure 3 shows the logical implementation of generation for inputs-outputs pairs.

▶ **Example 6.** For the Michelson program in Listing 1, two inputs arguments are both integers, we are able to use our sampler tool to generate its inputs/outputs relationships and also calculate its gas consumed. An example of results of 10 pairs inputs/outputs is showed in Listing 4, and gas consumed is **10.875**.

3 Synthesis : Search Process

In this section, we present how to rewrite and optimize the Michelson program in terms of preserved inputs-outputs relationships and its consumed gas. In actual work, the consumed gas is only used as the final judgment standard, and the consistency of the input and output relationship is a basic prerequisite for judging the qualification of the synthesized program. After these, we use Translation Validation (see Section 4) for our candidates to prove the programs are equivalent.

3.1 Well-typedness

The correct solutions have to be well-typed Michelson programs, while ill-typed programs may be generated during the search process. So we define the state of each node as `Well_Typed` or `Ill_Typed` and a type `full_node` (Listing 5) composed by this node and the state of this node. Based on the premise of the black box, the rewrite rules are the rules of randomly generating Michelson programs. Each node is randomly generated, which uses the program generated in the process is likely to be ill-typed. Nevertheless, it is very important to keep ill-typed nodes, because each node may be very close to our expected result.

Listing 4 Results


```
{ "samples": {
"0" : {
  "index": "0",
  "relation": {
    "input": { "parameter": "12", "storage_i": "-89" },
    "output": { "storage_o": "-47" }
  }
},
"1" : {
  "index": "1",
  "relation": {
    "input": { "parameter": "-125", "storage_i": "-150" },
    "output": { "storage_o": "-47" }
  }
},
"2" : {
  "index": "2",
  "relation": {
    "input": { "parameter": "-156", "storage_i": "-77" },
    "output": { "storage_o": "-47" }
  }
},

......

"9" : {
  "index": "9",
  "relation": {
    "input": { "parameter": "-4", "storage_i": "76" },
    "output": { "storage_o": "-47" }
  }
}
} }
```


Listing 5 type of full node (in OCaml)

```
type state = Well_Typed | Ill_Typed

type node = Michelson_value_sampler.node

type full_node = {n: node; st: state}
```

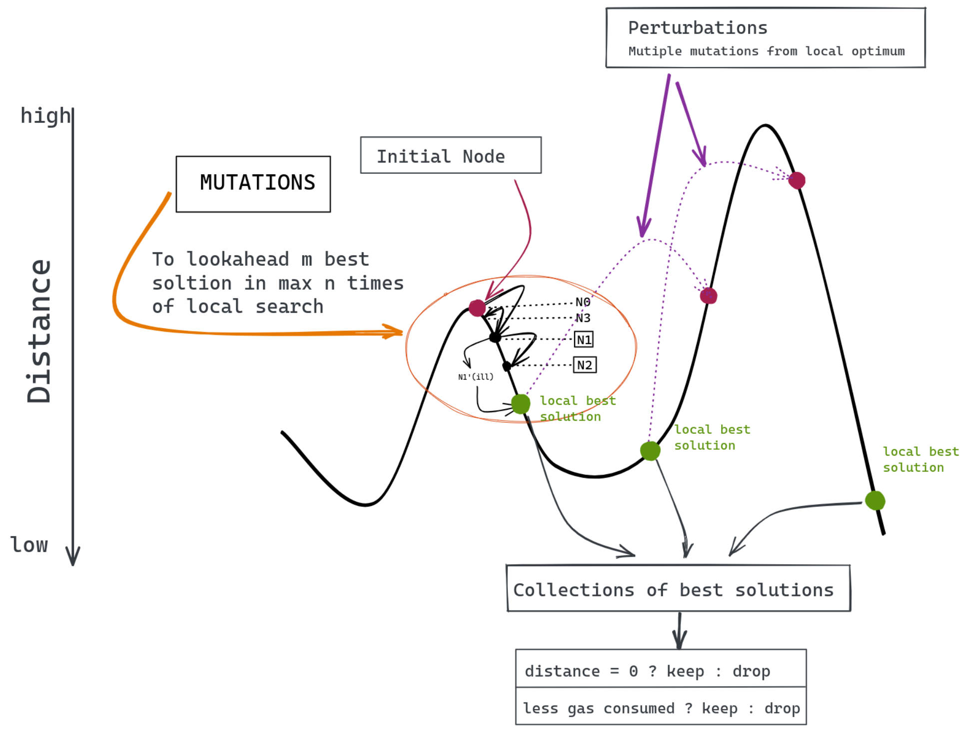

Figure 4 Rewriting Process

3.2 Random Rewrite Rules

We can interpret the process of rewriting as a search process. The termination signal of the search process comes from the satisfaction of the input-output relationships. The search space represented by the equivalent contract of a known Smart Contract is big, then S-metaheuristic would be the key to guide it to search faster and more effectively. As we have discussed, Iterated Local Search is implemented in our case.

Basically, we have two filters after the searching or rewriting. One is the input-output relationships, only the programs that match relationships can become the candidates. Then we need to check if we consume less gas. Candidates that consume less gas will be kept. The output should be a set of candidates of Michelson programs consuming less gas, as some of them might not be semantically equivalent with original one because of limited number of inputs-outputs pairs.

▶ Remark 7. Subsection 3.1 represents the importance of checking well-typedness of node. Thus at each step of mutation, we check the state of current node and score (see Subsection 3.2.1) it.

3.2.1 Scoring

Each program is scored. The most important element is the distance between the program generated and the original program. There are many different ways of computing such distance, i.e. edit distance and log-arithmetic distance. Edit distance is one good way of quantifying how dissimilar two strings are to one another by counting the minimum number of operations required to transform one string into the other. But in all my executions, considering that outputs of our basic blocks in programs are numbers, arithmetic distance is

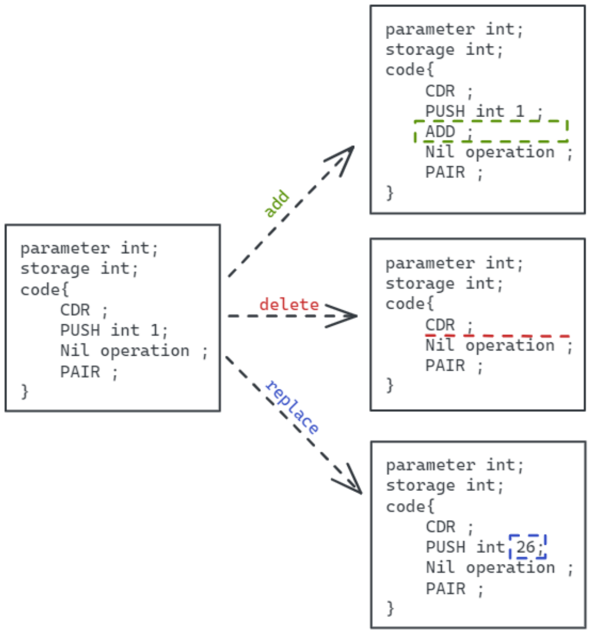

Figure 5 Modes of mutation

taken as the calculation approach (see Appendix A.3).

For each program generated in the process, we need to calculate sum of arithmetic distances between outputs, with the same input values. The exact process is to feed all input values of the samples of inputs-outputs relationships that has been obtained in sequence into the well-typed program, interpret the program, and calculate the distance between the actual output value and the expected output value. For the ill-typed program, due to its inexplicably and non-compilability, we designed a reasonable interval of positive integers and randomly selected a value as its scoring basis(i.e. distance). The design and control of this interval will be discussed in Section 3.2.4.

3.2.2 Mutation

The definition of mutation in this article is the process of randomly modifying one node to a new node. Basically, there are two mode: insertion and deletion. These two modes occur with equal possibility. Specially, we add one more mode with small possibility to be chosen, which is taken as replacement. The reason for the third mode is as follows. `PUSH` is one of important primitives to be inserted or deleted from nodes, and it has a same probability of generation as other primitives. It is also more special than others because the value pushed is a random integer(in this report). So for each primitive `PUSH` and each value, its probability of generation is much smaller. To compensate for this, we add this additional mode to replace the instruction of `PUSH *`.

For example, the demonstration in the Figure 5 is a basic mutation process. First, a tool that can randomly generate primitives within a limited range is established, then mutation mode will be chosen randomly from `Add`, `Delete`, `R_Push`. With high probability, this generated node will be randomly added or deleted at random position in the abstract semantic tree. It is also possible to replace the value to a random value after `PUSH` primitive. If there is no `PUSH` in the current node, it will randomly execution insertion or deletion.

Listing 6 Lookahead

```
...
let lookahead sol nbest max ctxt =
  let rec aux sol n max =
    match (n, max) with
    | (0, _) | (_, 0) -> return [sol]
    | (_, _) ->
        (*loop for local search*)
        loop sol 0 ctxt >>=? fun sol' ->
        if sol'.distance < sol.distance then
          aux sol' (n - 1) (max - 1) >>=? fun remain ->
          return (sol' :: remain)
          (*match remain with
            | [] -> Lwt.return [sol']
            | _ -> Lwt.return (sol' :: remain)*)
        else aux sol n (max - 1) >>=? fun remain ->
          return remain
  in
  aux sol nbest max
in
...
```

Local Search

Starting from the current node, the program can go through a one-step mutation to find a better program (closer distance) or a worse program (further distance). On this basis, starting from a node, it is allowed to "probe" a new node multiple times. At the same time, if a better node is found, that node will become a new starting point for detection and search for surrounding nodes(by multiple-mutation); all old distant nodes will be discarded. After enough attempts, only several best solutions will be retained. The above process is named *lookahead* (see Listing 6) in the program, which is also the critical part of Local Search.

After the end of each Local Search, we sort out the best nodes obtained and collect them into a large set to guide the subsequent iterations. At the same time, select the best well-typed node and judge whether it is possible to end a round of search (i.e. if the distance is 0, it means that a candidate is found, and this round of search ends).

3.2.3 Perturbation & Multiple-Rounds

The search process contains multiple rounds and each round contains multiple perturbations. Perturbation avoids local optimal results, multiple possible initial nodes are provided, thus local search is performed on different initial nodes.

As for the realization of perturbation, it is actually to execute multiple mutations randomly to the best local optimum node without detecting the content of the node, and then use it as the starting point of the Local Search.

The above process is implemented in one round of search, but in practice, one round of search may not be able to find qualified candidates, so we may need multiple rounds of search. The meaning of multiple rounds of search is not only to repeat the above process, but also to optimize the initial node, so that the generation of a new program does not start with a blank program, but the best well-typed program in the last search. Multiple rounds of search will improve the accuracy of program synthesis, but it also means that more time is consumed.

3.2.4 Design and Control Interval of Distance for Ill-typed Nodes

The explanation on why such a random number is chosen as the distance judgment of the ill-typed program is as follows. First of all, it is necessary to have a judgment mechanism. We need to guide the entire search process by comparing distances. The specific search process will be introduced in the next section. Secondly, during the search process it is hard to accurately judge the quality of an ill-typed program. It may or may not be a necessary part of the process of pointing to the expected result. Therefore, a given range of integers can give every node a chance to continue to evolve to a certain extent, until it finds a well-typed program and obtains an accurate distance value or it has a worse distances and is abandoned. Especially when the new well-typed program guided by ill-typed nodes has obvious better scores (closer distance), it shows that it has made very good progress after experiencing a lot of uninterpretable node evolution. The importance of distances of well-typed nodes is apparently higher than distances of ill-typed nodes, in terms of guide our search. This also means that without well-typed nodes, our search is purely random with no heuristic.

According to this analysis and multiple experiments of executions, we give several basic rules for the design of this interval. All these rules are implemented in the code [17] of this work.

1. The default initial bounds should have a high enough lower bound at least (also depending on the distance calculation method and the data size of the contract). Because our search starts from ill-typed node, if the random distance generated is too small, the search will lose the guide of scoring because it is hard to generate a well-typed node with a smaller distance at the beginning.
2. Except default bounds, lower bound and upper bound should be set around the best well-typed node so far. Also the probability of ill-typed node generated having a better distance should be controlled under a low level, because too easy to choose ill-typed nodes leads to less guide for search.
3. Every round of search should re-bound this interval based on the best solution found. Because if we keep two higher bounds, no ill-typed node will be kept.
4. Inside of each round, every perturbation allows to re-initialize bounds temporarily. When the best local optimum node is well-typed, we decrease the bounds according to this node. If not, we increase both the lower bound and the upper bound, to avoid missing better well-typed nodes.
5. Inside of each perturbation, every mutation doesn't change the bounds. Unless a new better well-typed node occurs, the bounds decrease according to this node.

3.3 Simplifying Candidates

It is obvious that there are some simple ways to simplify a program. For example, two consecutive `SWAP`s are a very simple deterministic rewriting rule: if we read a program that contains two consecutive `SWAP`s, we can directly modify the program and delete these two `SWAP`s. There are many other examples, we can add them to the rewrite rules. For example, `PUSH` operation followed by a `DROP`, `CDR` followed by a `DROP`, etc., can be replaced by simpler instructions. These definite rewriting rules are taken as an important means to simplify candidates that we found through the above random search process, and reduce gas consumption.

Listing 7 Candidates after search

```
### initial program ###
parameter int;
storage int;
code {DUP ; CDR; SWAP ; DROP ; DUP ; ADD;  NIL operation ; PAIR}
Cost : #consumed gas :10.2400000095

### solution ### N 0
{ parameter int ;
  storage int ;
  code { CDR ; DUP ; ADD ; NIL operation ; PAIR } }
State : Well-Typed
Cost : #consumed gas :7.13499999046
Distance : 0.

### solution ### N 1
{ parameter int ;
  storage int ;
  code { CDR ; DUP ; ADD ; NIL operation ; PAIR } }
State : Well-Typed
Cost : #consumed gas :7.13499999046
Distance : 0.
......
### solution ### N 40
{ parameter int ;
  storage int ;
  code { CDR ; DUP ; SWAP ; ADD ; NIL operation ; PAIR } }
State : Well-Typed
Cost : #consumed gas :7.92499995232
Distance : 0.
```

3.4 Implementation and Examples

We implement the process of exploring the graph generated by random rules, shown as Figure 4. Two main modules named `MUTATOR` and `RULES` contain most part of search process (see Appendix A.4).

Principles of search process

- Filter the candidates by preserving input-output relationships.
- Continue the exploration through the candidates by gas consumed.
- Stop when we cannot improve the score of the best candidate.
- Simplify candidates if we could.

Here is an example of candidates found by our search process in Listing7. The first two solutions are qualified candidates because they consume less gas, while the N°40 solution will be removed.

Listing 8 Type of stack element (in OCaml)

```
type sk_element =
  | Int of int
  | Pair of sk_element * sk_element
```

Listing 9 Sort of stack element and stack

```
let int_recognizer = stringsymbol "Int"

let pair_recognizer = stringsymbol "Pair"

let int_cstrct =
  Datatype.mk_constructor_s
    !ctxt
    "Int"
    int_recognizer
    [stringsymbol "int"]
    [Some int_sort]
    [1]

let pair_cstrct =
  Datatype.mk_constructor_s
    !ctxt
    "Pair"
    pair_recognizer
    [stringsymbol "car"; stringsymbol "cdr"]
    [None; None]
    [0; 0]

let sk_el_sort = Datatype.mk_sort_s !ctxt "sk_element"
    [int_cstrct; pair_cstrct]

let sk_sort = Z3List.mk_list_s !ctxt "stack" sk_el_sort
```

4 Translation Validation

So far, we generate programs that meet the requirements of input-output relations and gas consumption through a random process, but there is no guarantee that these programs is semantically consistent with our original program. Translation Validation is presented in this section.

4.1 Modeling Stacks and Encoding Instructions

We translate source program A and target program B into logical formulas. With the semantically equivalent input stacks SA and SB, use SMT solver to check if it is possible that output stacks from A and B differ. If the generated formula is satisfiable, they are not semantically equivalent.

A key element in our encoding is the representation of the stack and the elements it contains. In Z3, `sort` stands for a data type, and it has built-in integer sort and list sort, providing basic arithmetic methods. Based on this, we can process each stack into a Z3list of stack element sort. If we only consider *Integer* and *Pair* type of values in stack, we could

Listing 10 Examples of encoding instructions

```
...
    j  => s(0,j)
DROP(in j position):
    j + 1 => s(0,j+1) == tail s(0,j)
PUSH int 1:
    j + 2 => s(0,j+2) == cons ( Int 1 , s(0,j+1) )
SWAP :
    j + 3 => s(0,j+3) == cons ( head ( head s(0,j+2) ),
                               cons ( head s(0,j+2),
                                      tail (tail s(0,j+2)) ) )
...
```

define a data type (in OCaml) as Listing 8.

To implement this data type into Z3, we have to construct two constructors and create a new sort which stands for one stack element. And a stack sort would be a Z3list of stack element, as in Listing 9. By `accessors` defined in Z3, we are able to access the value of each `Expr` (i.e. General Expressions (terms) in Z3).

We aim to express a stack after executing j instructions with $j \in 0, ..., n\_ins$, where n_{ins} is the number of instructions. The expression of an abstractly defined stack type is treated as the initial stack before program execution, with symbolic expression `s(i,0)`, where i is the index of program. Obviously, for two different programs, we have the first restriction: the initial stack is equivalent, expressed as $s(0,0) == s(1,0)$.

Secondly, we encode instructions of Michelson program by establishing functions of effects of instructions in each Michelson program on the stack, and the execution process of the program is the modification process of the stack. By manipulation of the stack, we can finally get a stack as output states. The process of encoding is expressed as an example in Listing:10

To prove the equivalence of two Michelson programs, we need a second key constraint, that is, the output stack is inconsistent, expressed as $s(i,j) == s(i',j')$. Putting these two key constraints and the contraints built during the stack manipulation into Z3 SMT Solver, as long as the 'unsatisfiability' is obtained, we can consider the two programs to be semantically equivalent.

4.2 Examples

This subsection shows some examples of Translation Validation. It presents that semantic equivalence can be proved by the results of Z3 SMT Solver. In the first two examples (see Listing 11,12), we offer two pairs of Michelson programs and the tool returns 'unsatisfiability'. This means that for the same input stack, it is not possible to generate distinct output stacks after interpretations for each pair of programs. However in the third example (see Listing 13), that pair of programs is not semantic equivalent, thus the result of Translation Validation also confirms that.

With helps of Translation Validation, we are able to complete the last chain: the proof tool of this work.

Listing 11 Example 1

```
#### Program 0 ####
{parameter int ; storage int; code {DROP; PUSH int 20; PUSH int 2;
    PUSH int 3; DROP ; ADD ; NIL operation; PAIR}}
initial stack :s_0_0
stack - s_0_1
constraint - (= s_0_1 (tail s_0_0))
stack - s_0_2
constraint - (= s_0_2 (cons (Int 20) s_0_1))
stack - s_0_3
constraint - (= s_0_3 (cons (Int 2) s_0_2))
stack - s_0_4
constraint - (= s_0_4 (cons (Int 3) s_0_3))
stack - s_0_5
constraint - (= s_0_5 (tail s_0_4))
stack - s_0_6
constraint - (let ((a!1 (+ (int (head s_0_5))
                           (int (head (tail s_0_5))))))
  (= s_0_6 (cons (Int a!1) (tail (tail s_0_5)))))
final stack :s_0_6

#### Program 1 ####
{parameter int; storage int; code {DROP; PUSH int 18 ; PUSH int 4;
    ADD; NIL operation; PAIR}}
initial stack :s_1_0
stack - s_1_1
constraint - (= s_1_1 (tail s_1_0))
stack - s_1_2
constraint - (= s_1_2 (cons (Int 18) s_1_1))
stack - s_1_3
constraint - (= s_1_3 (cons (Int 4) s_1_2))
stack - s_1_4
constraint - (let ((a!1 (+ (int (head s_1_3))
                           (int (head (tail s_1_3))))))
  (= s_1_4 (cons (Int a!1) (tail (tail s_1_3)))))
final stack :s_1_4

#### Solver ####
(= s_0_0 s_1_0)
(distinct s_0_6 s_1_4)
(let ((a!1 (+ (int (head s_0_5)) (int (head (tail s_0_5))))))
  (= s_0_6 (cons (Int a!1) (tail (tail s_0_5)))))
(= s_0_5 (tail s_0_4))
(= s_0_4 (cons (Int 3) s_0_3))
(= s_0_3 (cons (Int 2) s_0_2))
(= s_0_2 (cons (Int 20) s_0_1))
(= s_0_1 (tail s_0_0))
(let ((a!1 (+ (int (head s_1_3)) (int (head (tail s_1_3))))))
  (= s_1_4 (cons (Int a!1) (tail (tail s_1_3)))))
(= s_1_3 (cons (Int 4) s_1_2))
(= s_1_2 (cons (Int 18) s_1_1))
(= s_1_1 (tail s_1_0))

unsatisfiable <------- means semantically equivalent
```

Listing 12 Example 2

```
#### Program 0 ####
{parameter int ; storage int; code {
       DUP ;
       CAR ;
       SWAP ;
       CDR ;
       PUSH int 20 ;
       ADD ;
       PUSH int 20 ;
       SUB ;
       ADD ;
       NIL operation;
       PAIR}}

#### Program 1 ####
{parameter int; storage int; code {
       DUP ;
       CDR ;
       SWAP ;
       CAR ;
       SUB ;
       NIL operation; PAIR}}

#### Solver ####
(= s_0_0 s_1_0)
(distinct s_0_9 s_1_5)
(let ((a!1 (+ (int (head s_0_8)) (int (head (tail s_0_8))))))
  (= s_0_9 (cons (Int a!1) (tail (tail s_0_8)))))
(let ((a!1 (- (int (head s_0_7)) (int (head (tail s_0_7))))))
  (= s_0_8 (cons (Int a!1) (tail (tail s_0_7)))))
(= s_0_7 (cons (Int 20) s_0_6))
(let ((a!1 (+ (int (head s_0_5)) (int (head (tail s_0_5))))))
  (= s_0_6 (cons (Int a!1) (tail (tail s_0_5)))))
(= s_0_5 (cons (Int 20) s_0_4))
(= s_0_4 (cons (cdr (head s_0_3)) (tail s_0_3)))
(let ((a!1 (cons (head (tail s_0_2))
                    (cons (head s_0_2) (tail (tail s_0_2))))))
  (= s_0_3 a!1))
(= s_0_2 (cons (car (head s_0_1)) (tail s_0_1)))
(= s_0_1 (cons (head s_0_0) s_0_0))
(let ((a!1 (- (int (head s_1_4)) (int (head (tail s_1_4))))))
  (= s_1_5 (cons (Int a!1) (tail (tail s_1_4)))))
(= s_1_4 (cons (car (head s_1_3)) (tail s_1_3)))
(let ((a!1 (cons (head (tail s_1_2))
                    (cons (head s_1_2) (tail (tail s_1_2))))))
  (= s_1_3 a!1))
(= s_1_2 (cons (cdr (head s_1_1)) (tail s_1_1)))
(= s_1_1 (cons (head s_1_0) s_1_0))

unsatisfiable <------- means semantically equivalent
```

Listing 13 Example 3

```
#### Program 0 ####
{parameter int ; storage int; code {DROP; PUSH int 20; PUSH int 2;
PUSH int 3; DROP ; ADD ; NIL operation; PAIR}}

#### Program 1 ####
{parameter int; storage int; code { PUSH int 18; DROP; DROP ;
PUSH int 4; NIL operation; PAIR}}

#### Solver ####
(= s_0_0 s_1_0)
(distinct s_0_6 s_1_4)
(let ((a!1 (+ (int (head s_0_5)) (int (head (tail s_0_5))))))
  (= s_0_6 (cons (Int a!1) (tail (tail s_0_5)))))
(= s_0_5 (tail s_0_4))
(= s_0_4 (cons (Int 3) s_0_3))
(= s_0_3 (cons (Int 2) s_0_2))
(= s_0_2 (cons (Int 20) s_0_1))
(= s_0_1 (tail s_0_0))
(= s_1_4 (cons (Int 4) s_1_3))
(= s_1_3 (tail s_1_2))
(= s_1_2 (tail s_1_1))
(= s_1_1 (cons (Int 18) s_1_0))

satisfiable <------- means semantically in-equivalent
```

5 Execution and Parameters

For the given example in Listing 1, we set all the parameters in Rewrite Rules (see Listing 14) and execute the program. Firstly, we generate 20 pairs of inputs-outputs relationships, all values are integers in [-255, 256], so the interval of possible distance is from -10220 to 10220. Thus we set the default bounds for an ill-typed node's distance as [25000, 50000] by the first rule in subsection 3.2.4. We set the max number of lookahead (defined in subsection 3.2.2), named *maxlookahead*, as 10, and the max loop times, named *n_mutations*), as 10, which means the maximal mutation in Local Search have 100 steps. We follow the rules (see subsection 3.2.4) defining the bounds of distance interval for ill-typed nodes, and keep a small probability for generating one ill-typed node with better distance. Thus what we set here is that upper bound is 100 times higher than lower bound (except the default initial bounds). In Local Search, every time we mutate a well-typed node, lower bounds of this interval is set to half of the historical best well-typed node. Every time of perturbation, we re-initialize bounds to the given one of each round.

The number of better nodes to search in lookahead process is set as 5, number of perturbation is set as 100 and maximum round is 5. Listing 15 shows the execution results we have in 25 minutes. In the first 15 minutes, we already find some qualified candidates, and it keeps searching for potential ones.

The gas consumed for original program is **10.875**, so we could have multiple qualified candidates with less gas consumed. Using Translation Validation, all candidates are checked (see Appendix B) and the best optimized program is **solution N°0** in Listing 15. It consumes only **7.29499983788** gas, saving around **32.92%** resources.

Listing 14 Parameters

```
let rewrite_random ~(update_runs : int) contract_file
    relationships_file initial_global_gas
    (module R : RULES) =
  let t = R.gen_init_node contract_file in
  ...
  let rec update ?heuristic_sol:(init_sol = init_sol)
      ?(bounds = R.default_bounds) ~(all_sols : R.sol list)
      ~(candidates : R.sol list) ~current_run:(run : int)
      (max_loop : int)
      (n_stop : int) =
    if run = max_loop || n_stop = 0 then Lwt.return candidates
    else (
      ...
      R.process
        ~bestlookahead:5
        ~maxlookahead:10
        ~n_pertubations:100
        ~n_mutations:10
        ~bounds
        ~init_sol
        relationships_file
        initial_global_gas
      >>= function
      ...
```

6 Conclusion and Future Work

We have presented a method for gas super-optimization of Smart Contracts based on S-metaheuristic in Tezos blockchain. Basically, our focus is on the stack operations for basic blocks of Michelson programs. This heuristics-based method offers one alternative way of superoptimizing Smart Contracts in terms of gas consumed. We build a basic tool in Tezos codebase. Currently, it can sample specific types of values and generate inputs-outputs relationships for a well-typed Michelson program; it searches programs by ILS algorithm and collect all possible candidates (i.e. consume less *gas* and qualify inputs-outputs relationships); it checks the semantic equivalence by Translation Validation between each candidate and original program, and returns optimized programs at the end.

Talking about its efficiency and quality, unfortunately there is no benchmark for now. Judging from only a few examples, its results are accurate and programs synthesized is truly optimized in terms of gas consumed. For simple Smart Contracts(e.g. with instructions less than 10 lines), this work has a good expectation to synthesize optimized programs after acceptable time of search. While there are some limitations of this tool. Firstly, for big Smart Contracts, we have to adjust parameters (including the interval bounds design and core parameters like number of rounds, number of `lookahead` execution times, etc.) and it would take much more time to search the space. The implementation of ILS algorithm in this work is still possible to be optimized. Secondly, this work considers only basic blocks of Michelson programs, which means there should be no control flow like for-loops or conditional control flow if we want to apply this tool. Thirdly, it lacks benchmarks to evaluate and improve this tool.

Future work should focus on efficiency and benchmarks. It should be able to be find

Listing 15 Execution Results

```
### solution ### N 0
{ parameter int ;
  storage int ;
  code { DROP ; PUSH int 47 ; NEG ; NIL operation ; PAIR } }
State : Well-Typed
Cost : #consumed gas :7.29499983788
Distance : 0.

### solution ### N 1
{ parameter int ;
  storage int ;
  code { DROP ; PUSH int 47 ; NEG ; NIL operation ; PAIR } }
State : Well-Typed
Cost : #consumed gas :7.29499983788
Distance : 0.

### solution ### N 2
{ parameter int ;
  storage int ;
  code { DROP ;
         PUSH int 47 ;
         ABS ;
         PUSH int -234 ;
         DROP ;
         NEG ;
         NIL operation ;
         PAIR } }
State : Well-Typed
Cost : #consumed gas :10.125
Distance : 0.

### solution ### N 3
{ parameter int ;
  storage int ;
  code { DROP ;
         PUSH int 47 ;
         ABS ;
         PUSH int -234 ;
         DROP ;
         NEG ;
         NIL operation ;
         PAIR } }
State : Well-Typed
Cost : #consumed gas :10.125
Distance : 0.

### solution ### N 4
{ parameter int ;
  storage int ;
  code { CAR ; DROP ; PUSH int 47 ; NEG ; NIL operation ; PAIR } }
State : Well-Typed
Cost : #consumed gas :8.09500002861
Distance : 0.
```

the optimized program faster, with a better implemented S-metaheuristics method (e.g. Metropolis Hasting [13] and Simulated Annealing, etc.) or with a better design of parameters. Also, for scoring Michelson programs, arithmetic distance may not be good enough for more complex situations. It would perform better by combining multiple methods, e.g. a variety of types of edit distance, log-arithmetic distance, etc. This work has proved the feasibility of this approach to a certain extent. Optimizing the S-metaheuristics algorithms implemented and improving the search mechanism on the basis of the current work should finally get a more satisfactory and efficient optimizer for complex Michelson programs.

References

1 Micheline documentation. `https://tezos.gitlab.io/shell/micheline.html`.

2 Michelson documentation. `https://tezos.gitlab.io/active/michelson.html`.

3 Tezos documentation. `https://tezos.gitlab.io/`.

4 Z3 smt solver. `https://github.com/Z3prover/z3`.

5 Elvira Albert, Pablo Gordillo, Albert Rubio, and Maria A. Schett. Synthesis of super-optimized smart contracts using max-smt. In *Computer Aided Verification*, pages 177–200, 2020. `doi:10.1007/978-3-030-53288-8_10`.

6 Sorav Bansal and Alex Aiken. Automatic generation of peephole superoptimizers. volume 41, pages 394–403, 10 2006. `doi:10.1145/1168857.1168906`.

7 Pete Bettinger and Kevin Boston. Forest planning heuristics-current recommendations and research opportunities for s-metaheuristics. *Forests*, 8:476, 12 2017. `doi:10.3390/f8120476`.

8 Leonardo de Moura and Nikolaj Bjørner. Z3: an efficient smt solver. volume 4963, pages 337–340, 04 2008. `doi:10.1007/978-3-540-78800-3_24`.

9 Benjamin Goldberg, Lenore Zuck, and Clark Barrett. Into the loops: Practical issues in translation validation for optimizing compilers. *Electronic Notes in Theoretical Computer Science*, 132(1):53–71, 2005. Proceedings of the 3rd International Workshop on Compiler Optimization Meets Compiler Verification (COCV 2004). URL: `https://www.sciencedirect.com/science/article/pii/S157106610505005X`, `doi:https://doi.org/10.1016/j.entcs.2005.01.030`.

10 Helena R. Lourenço, Olivier C. Martin, and Thomas Stützle. *Iterated Local Search: Framework and Applications*, pages 363–397. Springer US, Boston, MA, 2010. `doi:10.1007/978-1-4419-1665-5_12`.

11 Julian Nagele and M. A. Schett. Blockchain superoptimizer. *ArXiv*, abs/2005.05912, 2020.

12 A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In Bernhard Steffen, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 151–166, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

13 Christian P. Robert. *The Metropolis–Hastings Algorithm*, pages 1–15. American Cancer Society, 2015. URL: `https://onlinelibrary.wiley.com/doi/abs/10.1002/9781118445112.stat07834`, `arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/9781118445112.stat07834`, `doi:https://doi.org/10.1002/9781118445112.stat07834`.

14 Eric Schkufza, Rahul Sharma, and A. Aiken. Stochastic superoptimization. *ArXiv*, abs/1211.0557, 2013.

15 El-Ghazali Talbi. *Metaheuristics: from design to implementation*, volume 74. John Wiley & Sons, 2009.

16 Miguel Terra-Neves, Nuno Machado, Ines Lynce, and Vasco Manquinho. Concurrency debugging with maxsmt. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33(01):1608–1616, Jul. 2019. URL: `https://ojs.aaai.org/index.php/AAAI/article/view/3976`, `doi:10.1609/aaai.v33i01.33011608`.

17 Yu Tianchi. Michelson optimization. `https://gitlab.com/nomadic-labs/tezos/-/tree/tianchi@michelson-optimization-merge-snoop-random/src/bin_optimization`, 2021.

A Important Modules

A.1 Sampler

Listing 16 Michelson_value_sampler Module

```
open Protocol
open Alpha_context

type 'a ty = 'a Script_typed_ir.ty

type ex_ty = Script_ir_translator.ex_ty

type ex_value = Ex_Value : 'a ty * 'a -> ex_value

val get_ast : string -> Script.expr

val parse_expression : ?check:bool -> string -> Micheline_parser.node

type location = int

type node = (location, Script.prim) Micheline.node

val gen : 'a ty -> 'a

val gen_exvalue : ex_ty -> ex_value

val gen_node : string ->
    (location, Michelson_v1_primitives.prim) Micheline.node

val gen_node_with_target_type :
  string ->
  ( Script.node * context,
    Environment.Error_monad.error Environment.Error_monad.trace )
  result
  Lwt.t

val print : string Micheline.canonical -> string

val print_prims :
  string ->
  (string,
  Environment.Error_monad.error Environment.Error_monad.trace) result
  Lwt.t

val to_string : node -> string
```

A.2 Interpreter

Listing 17 Michelson_interpreteur Module

```
open Protocol
open Alpha_context
open Michelson_v1_primitives
```

```
open Script_interpreter

val run_script :
  context ->
  ?step_constants:step_constants ->
  string ->
  ?entrypoint:string ->
  storage:string ->
  parameter:string ->
  unit ->
  (execution_result, error trace) result Lwt.t

val print : string Micheline.canonical -> string

val parse_string : string -> Script.expr

val print_expanded : prim Micheline.canonical -> unit

val interprete_script :
  (context -> 'a) ->
  string ->
  context ->
  string ->
  string ->
  ((string * string) * string * 'a, 'b) result Lwt.t
```

A.3 Scoring

Listing 18 Distance Module

```
module type Dist = sig
  val dist : string -> string -> float

  val is_zero : float -> bool
end

(* To make distance computing methods *)
let mk_arith () =
  ( module struct
    let dist x y = Stdlib.abs_float
            (Float.of_string x -. Float.of_string y)

    let is_zero x = abs_float x < 0.5
  end : Dist )

let mk_hamming () =
  ( module struct
    exception Length_Diff

    let dist x y =
      if String.length x != String.length y then raise Length_Diff
      else
        let len = String.length x in
        let rec aux dist i len =
```

```
            if i = len then dist
            else if x.[i] == y.[i] then aux dist (i + 1) len
            else aux (dist + 1) (i + 1) len
          in
          Float.of_int (aux 0 0 len)

      let is_zero x = abs_float x < 1.
    end : Dist )

 (* in case that there are multiple distances *)
let sum_dists ?maxindex dist arr arr' =
  let max = match maxindex with
    Some i -> i | None -> Array.length arr in
  let rec aux i =
    if i < max then dist arr.(i) arr'.(i) +. aux (i + 1) else 0.
  in
  aux 0
```

A.4 Search

Listing 19 MUTATOR Module

```
module type MUTATOR = sig
  type t = Add | Delete | R_Push

  type state = Well_Typed | Ill_Typed

  type full_node = { n : node; st : state }

  val self_init_prim : unit -> node

  val init_prim : prim -> prim -> node

  val st_to_string : state -> string

  val typecheck :
    node ->
    (Script_tc_errors.type_map * Alpha_context.t)
    Environment.Error_monad.tzresult
    Lwt.t

  (* mutation without typechecking *)
  val mutate : node -> node Environment.Error_monad.tzresult Lwt.t

  (* using typecheck to generate a node *)
  val mutate_2 : node ->
        full_node Environment.Error_monad.tzresult Lwt.t
end
```

Listing 20 RULES Module

```
module type RULES = sig
  type t = node

```

```
  module M : MUTATOR

  val check_types_2 : t -> bool

  val swap_1 : t -> int list

  val cut : t -> int -> int list -> t

  val gen_init_node : string -> t

  val interprete_random_node :
    sample array -> t -> Alpha_context.t ->
    int -> (string array * string) Lwt.t

  type bounds = { floor : float; ceil : float }

  val default_bounds : bounds

  type sol =
    { full_node : M.full_node;
      cost : string;
      distance : float;
      bounds : bounds
    }

  val distance_list : float list ref

  val process :
    ?bestlookahead:int ->
    ?maxlookahead:int ->
    ?n_pertubations:int ->
    ?n_mutations:int ->
    bounds:bounds ->
    init_sol:sol ->
    string ->
    int ->
    sol list Environment.Error_monad.tzresult Lwt.t

  val output_file : string -> string

  val save_sols : sol list -> string -> unit Lwt.t
end
```

B Translation Validation

Here are the execution results of Translation Validation for distinct solutions in Listing 15.

Listing 21 Solution N 0

```
#### Program 0 ####
{parameter int;storage int; code { DROP ; PUSH int 47; NEG ;
    PUSH int 84 ; SWAP; SWAP ; DROP ; NIL operation ; PAIR}}
initial stack :s_0_0
stack - s_0_1
```

```
constraint - (= s_0_1 (tail s_0_0))
stack - s_0_2
constraint - (= s_0_2 (cons (Int 47) s_0_1))
stack - s_0_3
constraint - (let ((a!1 (Int (- 0 (int (head s_0_2))))))
  (= s_0_3 (cons a!1 (tail s_0_2))))
stack - s_0_4
constraint - (= s_0_4 (cons (Int 84) s_0_3))
stack - s_0_5
constraint - (let ((a!1 (cons (head (tail s_0_4))
                              (cons (head s_0_4)
                                    (tail (tail s_0_4))))))
  (= s_0_5 a!1))
stack - s_0_6
constraint - (let ((a!1 (cons (head (tail s_0_5))
                              (cons (head s_0_5)
                                    (tail (tail s_0_5))))))
  (= s_0_6 a!1))
stack - s_0_7
constraint - (= s_0_7 (tail s_0_6))
final stack :s_0_7

#### Program 1 ####
{ parameter int ;
  storage int ;
  code { DROP ; PUSH int 47 ; NEG ; NIL operation ; PAIR } }
initial stack :s_1_0
stack - s_1_1
constraint - (= s_1_1 (tail s_1_0))
stack - s_1_2
constraint - (= s_1_2 (cons (Int 47) s_1_1))
stack - s_1_3
constraint - (let ((a!1 (Int (- 0 (int (head s_1_2))))))
  (= s_1_3 (cons a!1 (tail s_1_2))))
final stack :s_1_3

#### Solver ####
(= s_0_0 s_1_0)
(distinct s_0_7 s_1_3)
(= s_0_7 (tail s_0_6))
(let ((a!1 (cons (head (tail s_0_5))
                 (cons (head s_0_5) (tail (tail s_0_5))))))
  (= s_0_6 a!1))
(let ((a!1 (cons (head (tail s_0_4))
                 (cons (head s_0_4) (tail (tail s_0_4))))))
  (= s_0_5 a!1))
(= s_0_4 (cons (Int 84) s_0_3))
(let ((a!1 (Int (- 0 (int (head s_0_2))))))
  (= s_0_3 (cons a!1 (tail s_0_2))))
(= s_0_2 (cons (Int 47) s_0_1))
(= s_0_1 (tail s_0_0))
(let ((a!1 (Int (- 0 (int (head s_1_2))))))
  (= s_1_3 (cons a!1 (tail s_1_2))))
(= s_1_2 (cons (Int 47) s_1_1))
(= s_1_1 (tail s_1_0))
```

```
unsatisfiable <------- means semantically equivalent
```

Listing 22 Solution N 2 (with same Program 0 in Listing 21)

```
#### Program 1 ####
{ parameter int ;
  storage int ;
  code { DROP ;
         PUSH int 47 ;
         ABS ;
         PUSH int -234 ;
         DROP ;
         NEG ;
         NIL operation ;
         PAIR } }
initial stack :s_1_0
stack - s_1_1
constraint - (= s_1_1 (tail s_1_0))
stack - s_1_2
constraint - (= s_1_2 (cons (Int 47) s_1_1))
stack - s_1_3
constraint - (let ((a!1 (ite (> (int (head s_1_2)) 0)
                (int (head s_1_2))
                (- 0 (int (head s_1_2))))))
  (= s_1_3 (cons (Int a!1) (tail s_1_2))))
stack - s_1_4
constraint - (= s_1_4 (cons (Int (- 234)) s_1_3))
stack - s_1_5
constraint - (= s_1_5 (tail s_1_4))
stack - s_1_6
constraint - (let ((a!1 (Int (- 0 (int (head s_1_5))))))
  (= s_1_6 (cons a!1 (tail s_1_5))))
final stack :s_1_6

#### Solver ####
(= s_0_0 s_1_0)
(distinct s_0_7 s_1_6)
(= s_0_7 (tail s_0_6))
(let ((a!1 (cons (head (tail s_0_5))
                 (cons (head s_0_5) (tail (tail s_0_5))))))
  (= s_0_6 a!1))
(let ((a!1 (cons (head (tail s_0_4))
                 (cons (head s_0_4) (tail (tail s_0_4))))))
  (= s_0_5 a!1))
(= s_0_4 (cons (Int 84) s_0_3))
(let ((a!1 (Int (- 0 (int (head s_0_2))))))
  (= s_0_3 (cons a!1 (tail s_0_2))))
(= s_0_2 (cons (Int 47) s_0_1))
(= s_0_1 (tail s_0_0))
(let ((a!1 (Int (- 0 (int (head s_1_5))))))
  (= s_1_6 (cons a!1 (tail s_1_5))))
(= s_1_5 (tail s_1_4))
(= s_1_4 (cons (Int (- 234)) s_1_3))
(let ((a!1 (ite (> (int (head s_1_2)) 0)
```

```
                (int (head s_1_2))
                (- 0 (int (head s_1_2))))))
  (= s_1_3 (cons (Int a!1) (tail s_1_2))))
(= s_1_2 (cons (Int 47) s_1_1))
(= s_1_1 (tail s_1_0))

unsatisfiable <------- means semantically equivalent
```

Listing 23 Solution N 4 (with same Program 0 in Listing 21)

```
#### Program 1 ####
{ parameter int ;
  storage int ;
  code { CAR ; DROP ; PUSH int 47 ; NEG ; NIL operation ; PAIR } }
initial stack :s_1_0
stack - s_1_1
constraint - (= s_1_1 (cons (car (head s_1_0)) (tail s_1_0)))
stack - s_1_2
constraint - (= s_1_2 (tail s_1_1))
stack - s_1_3
constraint - (= s_1_3 (cons (Int 47) s_1_2))
stack - s_1_4
constraint - (let ((a!1 (Int (- 0 (int (head s_1_3))))))
  (= s_1_4 (cons a!1 (tail s_1_3))))
final stack :s_1_4

#### Solver ####
(= s_0_0 s_1_0)
(distinct s_0_7 s_1_4)
(= s_0_7 (tail s_0_6))
(let ((a!1 (cons (head (tail s_0_5))
                 (cons (head s_0_5) (tail (tail s_0_5))))))
  (= s_0_6 a!1))
(let ((a!1 (cons (head (tail s_0_4))
                 (cons (head s_0_4) (tail (tail s_0_4))))))
  (= s_0_5 a!1))
(= s_0_4 (cons (Int 84) s_0_3))
(let ((a!1 (Int (- 0 (int (head s_0_2))))))
  (= s_0_3 (cons a!1 (tail s_0_2))))
(= s_0_2 (cons (Int 47) s_0_1))
(= s_0_1 (tail s_0_0))
(let ((a!1 (Int (- 0 (int (head s_1_3))))))
  (= s_1_4 (cons a!1 (tail s_1_3))))
(= s_1_3 (cons (Int 47) s_1_2))
(= s_1_2 (tail s_1_1))
(= s_1_1 (cons (car (head s_1_0)) (tail s_1_0)))

unsatisfiable <------- means semantically equivalent
```