# Exact and Linear-Time Gas-Cost Analysis

Ankush Das[1] and Shaz Qadeer[2]

[1] Carnegie Mellon University, USA
[2] Facebook, USA

**Abstract.** Blockchains support execution of smart contracts: programs encoding complex transaction protocols between distrusting parties. Due to their distributed nature, blockchains rely on third-party miners to execute and validate transactions. Miners are compensated by charging users with gas based on the execution cost of the transaction. To compute the exact gas cost, blockchains track gas cost dynamically creating its own overhead. This paper presents a static exact gas-cost analysis technique that can be employed to eliminate dynamic gas tracking. This approach presents further benefits such as providing miners with a trusted gas bound that can be verified in linear time, and eliminating out-of-gas exceptions. To handle recursion and unbounded computation, we propose a novel amortization technique that stores gas inside data structures. We have implemented our analysis technique in a tool called GasBoX which is evaluated on 13 standard smart contracts. Our soundness theorem proves that the gas bound verified by GasBoX exactly matches the gas cost at runtime and no dynamic gas tracking is necessary.

**Keywords:** Blockchains · Smart Contracts · Resource Analysis.

## 1 Introduction

Blockchains such as Ethereum [42] and Libra [8] allow execution of complex protocols between mutually distrusting parties through *smart contracts*. Smart contracts are programs typically written in a high-level language such as Solidity [16], Move [11] or Nomos [17] and compiled down to bytecode for execution on a distributed virtual machine. Smart contracts offer *transactions* (functions) that can be issued (called) by users to enforce such protocols, e.g. bidding in an auction, voting in an election, etc. Due to the distributed nature of blockchains, transactions are recorded by a large number of third-party entities, or *miners* (aka nodes) who are responsible for its execution. To prevent wastage of miner resources and compensate miners for their effort, users are charged for the execution cost of their transaction in the form of *gas units*.

Gas is the fuel of computation on blockchains. A *cost model* assigns a fixed gas cost to each operation. Gas cost of a transaction is the sum of the gas cost of each operation executed during the transaction. Users are responsible for providing a sufficient *gas limit* along with the transaction to cover the execution cost. If a user fails to provide sufficient gas, the transaction fails and all gas is lost! The user then has to re-issue the transaction with a higher gas limit. Since users need to be aware of execution cost prior to issuing a transaction, there is a wide variety of analysis tools [6, 5, 17, 23] to statically compute an *upper bound* on gas cost of transactions.

Unfortunately, upper gas bounds are inadequate. At runtime, if a user provides excess gas units, the leftover gas needs to be returned to the user. Thus, in existing blockchains such as Ethereum and Libra, a monitor function known as *dynamic gas meter* tracks the gas cost during execution. If the execution runs out of gas, the meter raises an *out-of-gas exception*, otherwise it returns the excess gas back to the user. Thus, despite the benefits of static gas analysis, blockchains still need to meter gas at runtime. Moreover, dynamic gas metering has its own limitations. First, it creates an execution overhead, inadvertently increasing the transaction gas cost. For the Libra blockchain, this overhead is about 20% of execution time [8]. Second, if the transaction runs out of gas, it does not provide any feedback to the user for transaction resubmission.

Upper gas bounds can also be unfair to miners. Miners are usually paid in proportion to the gas cost of a transaction. As a result, they often accept transactions with a high gas limit, hoping that transactions with a high gas limit will have a high gas cost. However, a malicious entity can trick this system by submitting transactions with a high gas limit but a low gas cost. Miners would accept such transactions only to discover that their compensation would be low and most of the gas is returned back to the user. Thus, there is a need to *provide miners with a trusted exact gas bound that can be verified efficiently* before accepting transactions.

In response, this article describes a static analysis technique with two goals: *(i) exact* gas analysis to eliminate dynamic metering, and *(ii) efficient* analysis that can be employed by miners. These goals pose unique challenges, particularly in the blockchain domain. The gas cost of a transaction can not only depend on its arguments, but also on global state, i.e., data structures already published on the blockchain. This global state can also potentially be modified by other transactions. Since a user has no control over when their transactions are actually mined, they cannot exactly determine the global state during execution. Verifying exact bounds can further be challenging in the presence of branching since the gas cost may vary along different branches.

To this end, blockchains recommend implementing contracts and transactions in a way that the gas cost does not depend on global state. Realizing this, our analysis tool only verifies gas bounds that are a *constant*, i.e., do not depend on either the arguments or the global state. As a result, our analysis is very efficient, and is *linear-time* in the size of the program and thus, can be employed by miners with minimal overhead. This overhead is further compensated since the virtual machine no longer needs to meter gas at runtime.

To compute exact bounds in the presence of branching, we need to ensure that branches have equal gas cost. We establish this by introducing a special operation `Gas.deposit`$(n)$ which deposits $n$ gas units in the transaction sender's account at runtime. We augment the less costly branch with such an expression with $n$ being the difference in the gas cost of both branches. We further illustrate that this mechanism is sufficient to produce exact gas bounds and eliminates the need for gas metering, improving the overall hygiene of the virtual machine.

To handle unbounded computation such as recursion and iteration over data structures like maps, we utilize *amortization* [39, 29, 27, 12]. We introduce $\texttt{Gas}(n)$ as a *first-class type* in the language to represent values with $n$ gas units which can then be stored inside data structures. During a transaction, this stored gas can be consumed to pay for the transaction cost. Thus, users pay in advance while building up such data structures and later, iteration would effectively pay for itself. Thus, such transactions have a constant static gas bound which are automatically verified by our analysis. We demonstrate that this amortization simplifies our gas analysis, prevents out-of-gas exceptions, and leads to more equitable gas-distribution schemes.

Although we have focused on constant gas bounds in this work, our analysis framework is general. In particular, the idea of depositing gas in sender's account to obtain exact gas bounds would still be applicable. The expressivity of the gas bounds can be enhanced by utilizing more sophisticated underlying logics, such as linear arithmetic [20] or SMT solvers [34]. However, such logics have a high computational complexity which would make the analysis inefficient, hampering its utility to miners. Although constant gas analysis precludes transactions that copy unbounded data structures such as lists and maps, we demonstrate that our tool can still analyze a large class of smart contracts.

We have implemented our analysis technique in a tool called GasBoX (GAS BOund eXact). The tool takes a function and an initial gas bound as input and either verifies that the bound is exact, or returns the program location where the virtual machine would run out of gas. Our analysis framework is *compositional*, thereby efficiently analyzing functions in isolation. We have designed a simplistic programming language modeled on Move [11] to illustrate the analysis technique and tool. We conducted 13 case studies implementing standard smart contracts such as auctions, elections, bank accounts, tokens, etc. and verified their gas bound using GasBoX. To the best of our knowledge, this is the first tool to compute exact gas bounds for smart contracts.

Overall, we make the following technical contributions:

1. design of a linear-time and exact gas-analysis technique for smart contracts
2. introduction of a novel deposit operation to avoid gas metering
3. gas amortization to handle unbounded computation
4. implementation of an analysis tool and case study on standard contracts

## 2 Overview of Gas Analysis

The static gas-cost analysis is realized using a Hoare logic style reasoning with an abstract notion of a static *gas tank*. This gas tank symbolically represents the amount of gas available to the execution engine at a program location, and is denoted using a natural number. For a program expression $e$, we follow the rule

$$\{\mathit{tank} = \phi + \mathcal{C}(e)\}\ e\ \{\mathit{tank} = \phi \mid \phi \geq 0\}$$

Here, $\phi+\mathcal{C}(e)$ represents the initial value of the gas tank, and $\mathcal{C}(e)$ denotes the gas cost of expression $e$. The rule states that if the gas tank value is $\phi+\mathcal{C}(e)$ before execution, then the gas tank value after execution is $\phi$. Our analysis is naturally compositional since gas cost is additive: the gas cost for $e \,;\, e'$ is $\mathcal{C}(e)+\mathcal{C}(e')$.

$$\frac{\begin{array}{c}\{tank = \phi + \mathcal{C}(e) + \mathcal{C}(e')\}\ e\ \{tank = \phi + \mathcal{C}(e') \mid \phi + \mathcal{C}(e') \geq 0\} \\ \{tank = \phi + \mathcal{C}(e')\}\ e'\ \{tank = \phi \mid \phi \geq 0\}\end{array}}{\{tank = \phi + \mathcal{C}(e) + \mathcal{C}(e')\}\ e \,;\, e'\ \{tank = \phi \mid \phi \geq 0\}}$$

## 2.1 Exact Bound Analysis and Runtime Overhead

We demonstrate our approach for exact gas analysis using an auction contract. Consider a function `addBid` which takes two arguments, `bidmap`: a reference to a map storing bids indexed by the address of their bidder, and `b`: a new bid to be added to the map represented using a `Coin` type.

```
fn addBid(bidmap: &Map<address, Coin>, b: Coin) {
  1. let bidder = GetTxnSenderAddress();
  2. if (Map.exists(copy(bidmap), copy(bidder))) then
  3.   tick($\mathcal{C}_{\texttt{MoveToAddr}}$); MoveToAddr(move(bidder), move(b));
  4. else
  5.   tick($\mathcal{C}_{\texttt{MapInsert}}$); Map.insert(move(bidmap), move(bidder), move(b));}
```

First, the bidder's address is computed and stored in the variable `bidder` (line 1). If `bidder` exists in the `bidmap` dictionary (line 2), then the bid is returned back to the bidder using the built-in `MoveToAddr` function (line 3). Otherwise, the bid is added to `bidmap` indexed by the bidder's address (line 5). For brevity, we allow a bidder to place a bid only once in this auction. Here, `move`$(v)$ moves the variable $v$ out of scope by passing it to the callee while `copy`$(v)$ creates a fresh *deep* copy of $v$. This distinction is necessary from the gas analysis perspective, since the gas cost of `move`$(v)$ can be statically determined, while the cost of `copy`$(v)$ depends on the size of $v$ (more details at the end of Section 2.2).

Gas cost of a function is defined w.r.t. a *cost model.* A cost model assigns a gas cost to each primitive operation. We simplify the analysis here by using the `tick` metric, which assigns a cost of $n$ to `tick`$(n)$, and 0 to all other operations. Statically, our analysis follows the rule

$$\{tank = \phi + n\}\ \texttt{tick}(n)\ \{tank = \phi \mid \phi \geq 0\}$$

In the `addBid` function above, we have only instrumented the `MoveToAddr` and `Map.insert` functions with ticks for simplicity of exposition. In practice, our implementation takes a cost model as input, and inserts `tick` for all operations automatically (explained in Section 3.1) so its burden does not fall on the programmer. With this model, the gas cost of `addBid` is $\mathcal{C}_{\texttt{MoveToAddr}}$ in the `then` branch and $\mathcal{C}_{\texttt{MapInsert}}$ in the `else` branch. Since we cannot statically determine which branch would be taken at runtime, the worst-case gas bound of `addBid` is $\texttt{max}(\mathcal{C}_{\texttt{MapInsert}}, \mathcal{C}_{\texttt{MoveToAddr}})$.

Since the statically derived gas bound is overapproximate, we need to dynamically meter the gas at runtime. Therefore, *despite the benefits of static* *gas analysis, we incur the overhead of metering the gas at runtime.* The gas meter will be responsible for returning the leftover gas back to the user at the end of execution. For the `addBid` function, if the initial provided gas is $\texttt{max}(\mathcal{C}_{\texttt{MapInsert}}, \mathcal{C}_{\texttt{MoveToAddr}})$, the leftover gas at the end of execution would be 0 or $\texttt{max}(\mathcal{C}_{\texttt{MapInsert}}, \mathcal{C}_{\texttt{MoveToAddr}}) - \texttt{min}(\mathcal{C}_{\texttt{MapInsert}}, \mathcal{C}_{\texttt{MoveToAddr}})$, depending upon which branch is executed.

To avoid dynamic metering, we need to compute an *exact* gas bound. To achieve this, we mandate that both branches have equal gas cost. To ensure this, we introduce an expression $\texttt{Gas.deposit}(n)$. Statically, the gas cost of this expression is $n$. Dynamically, executing this deposits $n$ units of gas in the account of the user who issued the transaction. The corresponding analysis rule is

$$\{tank = \phi + n\}\ \texttt{Gas.deposit}(n)\ \{tank = \phi \mid \phi \geq 0\}$$

Reimplementing the `addBid` function,

```
fn [$\mathcal{C}_{\texttt{MapInsert}} + \mathcal{C}_{\texttt{MoveToAddr}}$] addBid(bidmap: &Map<address, Coin>, b: Coin) {
  1. let bidder = GetTxnSenderAddress();
  2. if (Map.exists(copy(bidmap), copy(bidder))) then
       $\{tank = \mathcal{C}_{\texttt{MapInsert}} + \mathcal{C}_{\texttt{MoveToAddr}}\}$
  3.   tick($\mathcal{C}_{\texttt{MoveToAddr}}$); MoveToAddr(move(bidder), move(b));
       $\{tank = \mathcal{C}_{\texttt{MapInsert}} + \mathcal{C}_{\texttt{MoveToAddr}} - \mathcal{C}_{\texttt{MoveToAddr}} = \mathcal{C}_{\texttt{MapInsert}}\}$
  4.   Gas.deposit($\mathcal{C}_{\texttt{MapInsert}}$);
       $\{tank = \mathcal{C}_{\texttt{MapInsert}} - \mathcal{C}_{\texttt{MapInsert}} = 0\}$
  5. else
       $\{tank = \mathcal{C}_{\texttt{MapInsert}} + \mathcal{C}_{\texttt{MoveToAddr}}\}$
  6.   tick($\mathcal{C}_{\texttt{MapInsert}}$); Map.insert(move(bidmap), move(bidder), move(b));
       $\{tank = \mathcal{C}_{\texttt{MapInsert}} + \mathcal{C}_{\texttt{MoveToAddr}} - \mathcal{C}_{\texttt{MapInsert}} = \mathcal{C}_{\texttt{MoveToAddr}}\}$
  7.   Gas.deposit($\mathcal{C}_{\texttt{MoveToAddr}}$); }
       $\{tank = \mathcal{C}_{\texttt{MoveToAddr}} - \mathcal{C}_{\texttt{MoveToAddr}} = 0\}$
```

We have added the expression $\texttt{Gas.deposit}(\mathcal{C}_{\texttt{MapInsert}})$ in the `then` branch (line 4) and $\texttt{Gas.deposit}(\mathcal{C}_{\texttt{MoveToAddr}})$ in the `else` branch (line 7). With this addition, the gas cost of both branches becomes equal to $\mathcal{C}_{\texttt{MapInsert}} + \mathcal{C}_{\texttt{MoveToAddr}}$ as verified by the analysis (in blue). Since the gas tank value at the end of both branches is 0, we know that the *exact* gas bound of the `addBid` function is $\mathcal{C}_{\texttt{MapInsert}} + \mathcal{C}_{\texttt{MoveToAddr}}$ (described in blue along with the function declaration at the top).

The analysis takes the initial gas bound and the function definition as input and either verifies that the gas bound is exact or identifies the location where the execution will run out of gas. Intuitively, if $\phi \geq 0$ at each program location during the analysis, the gas bound is sufficient. Otherwise, the first location where $\phi < 0$ is the point where the execution runs out of gas. Moreover, the gas bound is exact if $\phi = 0$ after the `return` expression(s) in the function body.

***Advantages.*** Our analysis tool verifies the exact gas bound automatically. The soundness of our analysis proves that if a user supplies this gas bound with a transaction, *there is no need for dynamic metering.* The `Gas.deposit` expression ensures that the user does not lose any gas units; leftover gas is safely returned to the user. Our analysis tool automatically instruments the program with the

Gas.deposit operations, so its burden does not fall on the programmer. Furthermore, if the initial gas bound is not sufficient, the analysis identifies the program location where gas runs out, providing valuable feedback to the programmer.

One caveat here is that a programmer can still provide a high gas limit for a transaction and return most of the gas back to them using spurious Gas.deposit operations. To avoid this, we enforce that Gas.deposit operations are only inserted by our tool, and not by the programmer.

### 2.2 Handling Unbounded Computation

The auction contract also provides functionality for returning bids back to their respective bidders at the end of the auction. This is implemented with the recursive function below.

```
fn [$\mathcal{C}_{\texttt{MoveToAddr}} \cdot \texttt{sizeof(bidmap)}$] returnBids(bidmap : &Map<address, Coin>) {
  if (Map.size(copy(bidmap)) > 0) then
    $\{tank = \mathcal{C}_{\texttt{MoveToAddr}} \cdot \texttt{sizeof(bidmap)}\}$
    let (bidder, bid) = Map.remove_first(copy(bidmap)) ;
    $\{tank = \mathcal{C}_{\texttt{MoveToAddr}} \cdot (\texttt{sizeof(bidmap)} + 1)\}$
    tick($\mathcal{C}_{\texttt{MoveToAddr}}$) ; MoveToAddr(move(bidder), move(bid)) ;
    $\{tank = \mathcal{C}_{\texttt{MoveToAddr}} \cdot \texttt{sizeof(bidmap)}\}$
    returnBids(move(bidmap)) ; }
```

The function removes the first element from the map (remove_first), storing the key in bidder and value in bid. The function then calls MoveToAddr which transfers the bid into the bidder's account. Finally, the function recurses. Since we incur $\mathcal{C}_{\texttt{MoveToAddr}}$ cost for each recursive call (due to the $\texttt{tick}(\mathcal{C}_{\texttt{MoveToAddr}})$), the total cost of the returnBids function is $\mathcal{C}_{\texttt{MoveToAddr}} \cdot \texttt{sizeof}(\texttt{bidmap})$.

The analysis initiates with a gas tank value of $\mathcal{C}_{\texttt{MoveToAddr}} \cdot \texttt{sizeof}(\texttt{bidmap})$. The analysis then needs to verify that, in the else branch, $\texttt{sizeof}(\texttt{bidmap}) = 0$, thus the tank value is 0. In the then branch, the analysis needs to track that the size of bidmap decreases by 1 due to the $\texttt{remove\_first}()$ function, and the gas tank value decreases by $\mathcal{C}_{\texttt{MoveToAddr}}$ due to $\texttt{tick}(\mathcal{C}_{\texttt{MoveToAddr}})$. Thus, at the recursive call, we arrive at the invariant $\{tank = \mathcal{C}_{\texttt{MoveToAddr}} \cdot \texttt{sizeof}(\texttt{bidmap})\}$. To express and verify such invariants, the analysis would need to track the size of data structures and their relation to the gas tank value. If the control flow involves deeper nested loops and recursion, the gas bounds would involve non-linear expressions and the analysis would require sophisticated techniques to synthesize such invariants [4, 25, 21]. Furthermore, blockchains discourage non-constant gas cost transactions since they are vulnerable to out-of-gas exceptions and denial-of-service attacks [23].

***Gas Amortization.*** We instead propose a mechanism of *amortizing the linear cost* of returnBids over a series of bidding operations by *storing gas in data structures*. To pay for the gas cost of MoveToAddr, we store $\mathcal{C}_{\texttt{MoveToAddr}}$ units of gas with the bid in bidmap. This is defined using the type GasBid defined below.

```
resource GasBid {
  gas : Gas($\mathcal{C}_{\texttt{MoveToAddr}}$),       // $\mathcal{C}_{\texttt{MoveToAddr}}$ gas units stored inside GasBid
  bid : Coin }                 // stores bid to be placed in auction
```

Our language allows declaration of two kinds of types: *structs* and *resources*. They are both analogous to classes in object-oriented languages, except that they differ in their treatment. Objects of struct types represent functional data structures: they can be moved or copied, whereas objects of resource types represent assets: they cannot be copied, only moved; they are treated *linearly* [22].

We introduce a new primitive linear type in the language $\texttt{Gas}(n)$ where $n$ is a constant natural number. Statically, a variable $v : \texttt{Gas}(n)$ stores $n$ units of gas. Constructing a variable of type $\texttt{Gas}(n)$ consumes $n$ gas units from the gas tank, while destructing it produces $n$ gas units that are added to the gas tank. Formally, the introduction and elimination rules are described as

$$\{\mathit{tank} = \phi + n\}\ \texttt{Gas.construct}(n)\ \{\mathit{tank} = \phi \mid \phi \geq 0\}$$
$$\{\mathit{tank} = \phi \mid v : \texttt{Gas}(n)\}\ \texttt{Gas.destruct}(v)\ \{\mathit{tank} = \phi + n\}$$

***Amortized Auction.*** We reimplement the auction contract storing $\mathcal{C}_{\texttt{MoveToAddr}}$ gas units in the `GasBid` resource type. In this version, the bidder pays for the return of bids in advance.

```
fn [$\mathcal{C}_{\texttt{MapInsert}} + 2\mathcal{C}_{\texttt{MoveToAddr}}$] addBid(bidmap: &Map<address, GasBid>, b: Coin) {
  let bidder = GetTxnSenderAddress();
  if (Map.exists(copy(bidmap), copy(bidder))) then
    $\{\mathit{tank} = \mathcal{C}_{\texttt{MapInsert}} + 2\mathcal{C}_{\texttt{MoveToAddr}}\}$
    tick($\mathcal{C}_{\texttt{MoveToAddr}}$); MoveToAddr(move(bidder), move(b));
    $\{\mathit{tank} = \mathcal{C}_{\texttt{MapInsert}} + 2\mathcal{C}_{\texttt{MoveToAddr}} - \mathcal{C}_{\texttt{MoveToAddr}} = \mathcal{C}_{\texttt{MapInsert}} + \mathcal{C}_{\texttt{MoveToAddr}}\}$
    Gas.deposit($\mathcal{C}_{\texttt{MapInsert}} + \mathcal{C}_{\texttt{MoveToAddr}}$);
    $\{\mathit{tank} = \mathcal{C}_{\texttt{MapInsert}} + \mathcal{C}_{\texttt{MoveToAddr}} - \mathcal{C}_{\texttt{MapInsert}} - \mathcal{C}_{\texttt{MoveToAddr}} = 0\}$
  else
    $\{\mathit{tank} = \mathcal{C}_{\texttt{MapInsert}} + 2\mathcal{C}_{\texttt{MoveToAddr}}\}$
    let g = Gas.construct($\mathcal{C}_{\texttt{MoveToAddr}}$);
    $\{\mathit{tank} = \mathcal{C}_{\texttt{MapInsert}} + 2\mathcal{C}_{\texttt{MoveToAddr}} - \mathcal{C}_{\texttt{MoveToAddr}} = \mathcal{C}_{\texttt{MapInsert}} + \mathcal{C}_{\texttt{MoveToAddr}}\}$
    let gb = pack<GasBid> {gas: move(g), bid: move(b)};
    tick($\mathcal{C}_{\texttt{MapInsert}}$); Map.insert(move(bidmap), move(bidder), move(gb));
    $\{\mathit{tank} = \mathcal{C}_{\texttt{MapInsert}} + \mathcal{C}_{\texttt{MoveToAddr}} - \mathcal{C}_{\texttt{MapInsert}} = \mathcal{C}_{\texttt{MoveToAddr}}\}$
    Gas.deposit($\mathcal{C}_{\texttt{MoveToAddr}}$); }
    $\{\mathit{tank} = \mathcal{C}_{\texttt{MoveToAddr}} - \mathcal{C}_{\texttt{MoveToAddr}} = 0\}$
```

```
fn [$0$] returnBids(bidmap : &Map<address, GasBid>) {
  if (Map.size(copy(bidmap)) > 0) then
    let (bidder, gbid) = Map.remove_first(copy(bidmap)) ;
    let (g, bid) = unpack<GasBid>(move(gbid));
    $\{\mathit{tank} = 0 \mid \texttt{g} : \texttt{Gas}(\mathcal{C}_{\texttt{MoveToAddr}})\}$
    Gas.destruct(g);
    $\{\mathit{tank} = \mathcal{C}_{\texttt{MoveToAddr}}\}$
    tick($\mathcal{C}_{\texttt{MoveToAddr}}$) ; MoveToAddr(move(bidder), move(bid)) ;
    $\{\mathit{tank} = \mathcal{C}_{\texttt{MoveToAddr}} - \mathcal{C}_{\texttt{MoveToAddr}} = 0\}$
    returnBids(move(bidmap)) ; }
```

The `bidmap` argument to `addBid` now has type $\&\texttt{Map}\langle\texttt{address}, \texttt{GasBid}\rangle$. The `else` branch of `addBid` first constructs $\texttt{g} : \texttt{Gas}(\mathcal{C}_{\texttt{MoveToAddr}})$ and then uses `pack` to create $\texttt{gb} : \texttt{GasBid}$. The `pack` expression takes the value of each field of a resource (or struct) and creates an object of that type. The object `gb` is then inserted and the remaining gas is deposited. The `returnBids` function first unpacks `gbid` : `GasBid`, storing the gas and bid in the variables `g` and `bid`. The gas is then destructed to pay for the cost of `tick`($\mathcal{C}_{\texttt{MoveToAddr}}$).

The increased gas cost of `addBid` is $\mathcal{C}_{\texttt{MapInsert}} + 2\mathcal{C}_{\texttt{MoveToAddr}}$. Out of this, $\mathcal{C}_{\texttt{MapInsert}} + \mathcal{C}_{\texttt{MoveToAddr}}$ gas units are consumed for the cost of function execution, while $\mathcal{C}_{\texttt{MoveToAddr}}$ gas units are stored in `bidmap` for future use. The gas cost of `returnBids` is now 0. It consumes $\mathcal{C}_{\texttt{MoveToAddr}}$ gas units in every recursive call, which is provided by the gas stored inside `bidmap`.

***Advantages.*** The advantages of amortization by storing gas inside data structures are manifold. First, it simplifies the analysis that no longer needs to synthesize complicated invariants and track data structure sizes. Second, blockchains such as Libra [8] and Ethereum [42] assign a maximum gas limit to transactions. The gas cost of the unamortized `returnBids` function is $\mathcal{C}_{\texttt{MoveToAddr}} \cdot$ `sizeof(bidmap)`. This cost increases as the size of `bidmap` increases; if the size of `bidmap` increases beyond a threshold, the gas cost would *exceed the maximum gas limit* allowed for a transaction. The bids would then get stuck in the contract with no possibility of retrieving them [23]. Finally, this distribution of gas cost is more *equitable*. The bidders should be responsible for paying for both the cost of bidding as well as retrieving their bids from the auction. In the unamortized version, the user who issues `returnBids` bears the burden of paying for return of all the bids back to their respective bidders. The advantage of eliminating gas metering is also enhanced: the overhead of metering is linear in the execution time, while the overhead of static analysis is linear in the *program size*.

***Move vs Copy.*** The distinction between move and copy operations is crucial for our static gas-cost analysis. Semantically, `move`($v$) corresponds to a shallow copy of $v$ whose gas cost only depends on the type of $v$. On the other hand, `copy`($v$) corresponds to a deep copy of $v$, whose gas cost depends on the size of $v$. Since our analysis technique only handles constants, we disallow copy of unbounded data structures such as maps. Remarkably, we can analyze a large number of contracts despite this restriction (see Section 4) since we allow copy on primitive types and structs (and resources) containing them. Since we are working on an intermediate-level language, we require the move and copy operations to be explicit. However, they can be implicit in a source language, and be automatically inserted by a compiler, e.g. Move [11].

## 3 Formal Analysis

This section formalizes our source programming language, the static gas analysis and the formal gas semantics. We conclude with a soundness theorem connecting the static analysis with the semantics establishing that the gas bound verified by the static analysis is exactly matched at runtime.

### 3.1 A Simplistic Programming Language

Our language is modeled on Move [11], and provides an intuitive intermediate-level surface syntax on top of Move bytecode.

***Types.*** The language features standard primitive types such as `int` and `bool` representing integers and booleans, respectively. It also provides a built-in map data type $\texttt{Map}\langle\tau_1, \tau_2\rangle$ where $\tau_1$ and $\tau_2$ are the key and value types, respectively. In addition, multiple values (with different types) can be packed together using `struct` and `resource` types. Finally, the language provides basic support for references, providing type $\&\tau$ to refer values of type $\tau$. Although Move distinguishes mutable and immutable references, we consider all references as mutable since it is orthogonal to gas analysis. At runtime, references are represented by constant size addresses and do not pose additional challenges for gas analysis.

We also introduce $\texttt{Gas}(n)$ as a first-class type in our language, where $n$ is a constant natural number. This is used to store gas in data structures to share and amortize the gas cost of transactions, as demonstrated in Section 2. Thus, the type grammar for our language is

$$\tau \ ::= \ \texttt{int} \mid \texttt{bool} \mid \texttt{Map}\langle\tau,\tau\rangle \mid \&\,\tau \mid V \mid \texttt{Gas}(n)$$

$V$ represents type names, denoting struct and resource types (e.g. `GasBid`). The syntax for declaring structs and resources is described later (end of Section 3.1).

***Expressions.*** The expression language is expressed using the following grammar. Below, $n$ is a constant integer, while $v$ is a variable name.

$$\begin{aligned} e ::= {} & n \mid \texttt{true} \mid \texttt{false} \mid \ldots (*\text{ standard expressions for primitive types } *) \\ \mid {} & \texttt{pack}\langle\tau\rangle\{\texttt{f}_1 : e, \ldots, \texttt{f}_n : e\} \mid \texttt{unpack}\langle\tau\rangle(e) \mid \&v.\texttt{f} \mid \&v \\ \mid {} & \texttt{move}(v) \mid \texttt{copy}(v) \mid \texttt{g}(\overline{e}) \\ \mid {} & \texttt{let}\ \overline{v} = e \mid v \leftarrow e \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \mid e\ ;\ e \mid \texttt{return}\ e \\ \mid {} & \texttt{tick}(n) \mid \texttt{Gas.construct}(n) \mid \texttt{Gas.destruct}(v) \mid \texttt{Gas.deposit}(n) \end{aligned}$$

Our language features standard expressions for integer and boolean operations. These include binary arithmetic $(+, -, *, /)$, comparison $(>, \geq, <, \leq)$ and relational $(\&\&, ||)$ operators. Pack and unpack expressions are used to construct and destruct objects of struct (and resource) types, respectively. The expression $\texttt{pack}\langle\tau\rangle\{\texttt{f}_1 : e_1, \ldots, \texttt{f}_n : e_n\}$ packs together expressions $(e_1, \ldots, e_n)$ assigned to fields $\texttt{f}_1, \ldots, \texttt{f}_n$ respectively, and creates an object of type $\tau$. Dually, $\texttt{unpack}\langle\tau\rangle(e)$ destructs object $e : \tau$ and returns the tuple $(e_1, \ldots, e_n)$ corresponding to each field. Additionally, we can reference the field `f` of a variable $v$ using $\&v.\texttt{f}$. References of a variable $v$ can be taken using $\&v$. A variable $v$ can be moved or copied using $\texttt{move}(v)$ and $\texttt{copy}(v)$ respectively. Function calls have the usual syntax $\texttt{g}(e_1, \ldots, e_n)$ calling function `g` with argument expressions $e_1, \ldots, e_n$. We also provide standard map functions such as insertion, removal and checking size. Additionally, the function $\texttt{remove\_first}()$ removes and returns the first key-value pair in a map and is used to iterate over maps. The `let` expression evaluates $e$ and assigns its value to a set of fresh variables $\overline{v}$. We use a set of variables because expressions `unpack` and `remove_first` return multiple values. The value of variable $v$ is updated to the value of $e$ using $v \leftarrow e$. Branches are created with $\texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e$, executing $e_1$ or $e_2$ depending upon whether $e$ evaluates to `true` or `false` respectively. Expressions are composed using $e_1\ ;\ e_2$

and returned using $\texttt{return}\ e$. Finally, we provide blockchain-specific operations and functions (similar to Move), e.g., `GetTxnSenderAddress` and `MoveToAddr`. These blockchain-specific expressions have a constant gas cost, and do not pose additional challenges w.r.t. gas analysis.

***Cost Model and Gas Expressions.*** Our analysis needs to account for the gas cost assigned to each operation. We simplify the analysis by adding `tick` expressions [27, 19] based on a cost model that assigns a constant gas cost to each primitive operation. Our implementation then automatically instruments the program by adding ticks for each primitive operation based on the cost model. We describe the rules of instrumentation with the convention that $[\![e]\!]$ represents the instrumented version of $e$ (analogous cases skipped for brevity).

$$\begin{aligned}
[\![\texttt{pack}\langle\tau\rangle\{\texttt{f}_1 : e_1, \ldots\}]\!] &:= \texttt{tick}(\mathcal{C}_{\texttt{pack}} \cdot \texttt{size}(\tau))\ ;\ \texttt{pack}\langle\tau\rangle\{\texttt{f}_1 : [\![e_1]\!], \ldots\} \\
[\![\texttt{unpack}\langle\tau\rangle(e)]\!] &:= \texttt{tick}(\mathcal{C}_{\texttt{unpack}} \cdot \texttt{size}(\tau))\ ;\ \texttt{unpack}\langle\tau\rangle([\![e]\!]) \\
[\![\texttt{move}(v)]\!] &:= \texttt{tick}(\mathcal{C}_{\texttt{move}} \cdot \texttt{size}(\tau))\ ;\ \texttt{move}(v) \qquad (v : \tau) \\
[\![\texttt{g}(e_1, \ldots, e_n)]\!] &:= \texttt{tick}(\mathcal{C}_{\texttt{g}})\ ;\ \texttt{g}([\![e_1]\!], \ldots, [\![e_n]\!]) \\
[\![\texttt{let}\ v = e]\!] &:= \texttt{tick}(\mathcal{C}_{\texttt{let}})\ ;\ \texttt{let}\ v = [\![e]\!] \\
[\![v \leftarrow e]\!] &:= \texttt{tick}(\mathcal{C}_{\texttt{asgn}})\ ;\ [\![v]\!] \leftarrow [\![e]\!] \\
[\![\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2]\!] &:= \texttt{tick}(\mathcal{C}_{\texttt{if}})\ ;\ \texttt{if}\ [\![e]\!]\ \texttt{then}\ [\![e_1]\!]\ \texttt{else}\ [\![e_2]\!] \\
[\![e_1\ ;\ e_2]\!] &:= [\![e_1]\!]\ ;\ \texttt{tick}(\mathcal{C}_{\texttt{seq}})\ ;\ [\![e_2]\!] \\
[\![\texttt{return}\ e]\!] &:= \texttt{tick}(\mathcal{C}_{\texttt{ret}})\ ;\ \texttt{return}\ e
\end{aligned}$$

The costs $\mathcal{C}_{\texttt{i}}$'s above represent the cost model which we require the programmer to provide. The gas cost $\mathcal{C}_{\texttt{g}}$ of function `g` is determined from the declaration of `g` (described in the end of Section 3.1). The analysis is then completely *parametric in the cost model*, providing full flexibility to the programmer to specify their own cost model. The gas cost can also depend on $\texttt{size}(\tau)$, defined as

$$\begin{aligned}
&\texttt{size}(\texttt{int}) = 4 \qquad \texttt{size}(\texttt{bool}) = 2 \qquad \texttt{size}(\texttt{Gas}(n)) = 4 \qquad \texttt{size}(\&\tau) = 8 \\
&\texttt{size}(\texttt{Map}\langle\tau_1, \tau_2\rangle) = \texttt{size}(\tau_1) + \texttt{size}(\tau_2) \qquad \texttt{size}(V) = \Sigma_{i=1}^{n}\texttt{size}(\tau_i)
\end{aligned}$$

where $V$ denotes a struct or resource type, and $\tau_i$'s denote the type of its fields.

We provide special syntax for creating and destroying gas variables. A variable $v$ of type $\texttt{Gas}(n)$ (for a constant number $n$) can be constructed using $\texttt{Gas.construct}(n)$, while destructed using $\texttt{Gas.destruct}(v)$. We can further deposit gas in the sender's account with $\texttt{Gas.deposit}(n)$.

***Program.*** A program is a sequence of (possibly mutually) recursive type and function declarations. Their grammar is

$$\begin{aligned}
\langle \mathit{decl} \rangle ::= {} & \texttt{resource}\ V\ \{\texttt{f}_1 : \tau, \ldots, \texttt{f}_n : \tau\} \mid \texttt{struct}\ V\ \{\texttt{f}_1 : \tau, \ldots, \texttt{f}_n : \tau\} \\
\mid\ & \texttt{fn}\ [\mathcal{G}]\ F(v : \tau, \ldots, v : \tau) \to \tau\ \{e\}
\end{aligned}$$

Type declarations are used to define struct and resource types. The syntax $\texttt{resource}\ V\ \{\texttt{f}_1 : \tau_1, \ldots, \texttt{f}_n : \tau_n\}$ defines type $V$ with fields $\texttt{f}_1, \ldots, \texttt{f}_n$ (with corresponding types $\tau_1, \ldots, \tau_n$ respectively). Structs have a similar syntax. Functions are declared using $\texttt{fn}\ [\mathcal{G}]\ F(v_1 : \tau_1, \ldots, v_n : \tau_n) \to \tau\ \{e\}$ defines function

$F$ with $n$ arguments $v_1 : \tau_1, \ldots, v_n : \tau_n$, return type $\tau$, function body $e$ and gas bound $\mathcal{G}$ as a constant natural number. We store the definition of each type and function (with initial gas bound) in a *global signature* $\Sigma$. This signature $\Sigma$ is referenced during tick instrumentation to obtain the gas cost of each function call. Our analysis takes a program as input and verifies that $\mathcal{G}$ is an exact gas bound for each function $F$ in the program.

### 3.2 Static Gas Analysis

The analysis is formalized as a quantitative Hoare triple $\{\mathcal{G} \mid \Gamma\}\ e\ \{\mathcal{G}' \mid \Gamma'\}$. Here, $e$ denotes the expression that will be *gas-analyzed*; $\Gamma$ and $\Gamma'$ store the context (type of variables in scope) before and after the execution of $e$; $\mathcal{G}$ and $\mathcal{G}'$ track the gas tank value as a natural number before and after the execution of $e$, respectively. As a convention, we refer to $\mathcal{G}$ and $\Gamma$ as the *pre-gas* and *pre-context* together called *pre-state*, and $\mathcal{G}'$ and $\Gamma'$ as the *post-gas* and *post-context* of $e$ together called *post-state*, respectively. In the above judgment, there is an implicit invariant that $\mathcal{G}, \mathcal{G}' \geq 0$.

***Expressions.*** We describe selected rules that update the gas tank.

$$\frac{\mathcal{G} = \mathcal{G}' + n}{\{\mathcal{G} \mid \Gamma\}\ \texttt{Gas.construct}(n)\ \{\mathcal{G}' \mid \Gamma\}}\ \texttt{I}_{\texttt{gas}}$$

$$\frac{\mathcal{G}' = \mathcal{G} + n}{\{\mathcal{G} \mid \Gamma, v : \texttt{Gas}(n)\}\ \texttt{Gas.destruct}(v)\ \{\mathcal{G}' \mid \Gamma\}}\ \texttt{E}_{\texttt{gas}}$$

$$\frac{\mathcal{G} = \mathcal{G}' + n}{\{\mathcal{G} \mid \Gamma\}\ \texttt{Gas.deposit}(n)\ \{\mathcal{G}' \mid \Gamma\}}\ \texttt{D}_{\texttt{gas}}$$

Constructing a variable of type $\texttt{Gas}(n)$ consumes $n$ units of gas from the tank. Dually, $\texttt{Gas.destruct}(v)$ looks up the type of $v : \texttt{Gas}(n)$ in the context $\Gamma$ and adds $n$ gas units to the gas tank. The variable $v$ is then removed from $\Gamma$ since it is no longer in scope. $\texttt{Gas.deposit}(n)$ removes $n$ units of gas from the tank and deposits it in the user's account.

$$\frac{\mathcal{G} = \mathcal{G}' + n}{\{\mathcal{G} \mid \Gamma\}\ \texttt{tick}(n)\ \{\mathcal{G}' \mid \Gamma\}}\ \texttt{tick}$$

Executing $\texttt{tick}(n)$ consumes $n$ gas units.

$$\frac{\{\mathcal{G}_0 \mid \Gamma_0\}\ e_1\ \{\mathcal{G}_1 \mid \Gamma_1\} \qquad \ldots \qquad \{\mathcal{G}_{n-1} \mid \Gamma_{n-1}\}\ e_n\ \{\mathcal{G}_n \mid \Gamma_n\}}{\{\mathcal{G}_0 \mid \Gamma_0\}\ \texttt{pack}\langle\tau\rangle\{\texttt{f}_1 : e_1, \ldots, \texttt{f}_n : e_n\}\ \{\mathcal{G}_n \mid \Gamma_n\}}\ \texttt{pack}$$

Packing $n$ expressions $e_1 \ldots, e_n$ requires analyzing each expression and composing the gas tanks and contexts together. The post-state of $e_i$ becomes the pre-state for $e_{i+1}$. Unpacking an expression $e$ corresponds to gas-analyzing it.

$$\frac{\{\mathcal{G}_0 \mid \Gamma_0\}\ e_1\ \{\mathcal{G}_1 \mid \Gamma_1\} \qquad \ldots \qquad \{\mathcal{G}_{n-1} \mid \Gamma_{n-1}\}\ e_n\ \{\mathcal{G}_n \mid \Gamma_n\}}{\{\mathcal{G}_0 \mid \Gamma_0\}\ \texttt{g}(e_1, \ldots, e_n)\ \{\mathcal{G}_1 \mid \Gamma_1\}}\ \texttt{call}$$

For function calls, we analyze each argument, composing the gas tanks and contexts from left to right (similar to `pack`) since the expressions are evaluated from left to right at runtime. Note that there is no need to analyze the function body of `g` since the cost of calling and evaluating `g` is already accounted for by the tick instrumentation that inserts $\mathcal{C}_{\mathtt{g}}$ just before the function call. This observation is crucial to obtain a linear-time gas analysis.

$$\frac{\{\mathcal{G} \mid \Gamma\}\ e\ \{\mathcal{G}' \mid \Gamma'\} \qquad \Gamma \vdash e : \tau}{\{\mathcal{G} \mid \Gamma\}\ \mathtt{let}\ v = e\ \{\mathcal{G}' \mid \Gamma', v : \tau\}}\ \mathtt{let}$$

For `let` expressions, we use an auxiliary judgment: $\Gamma \vdash e : \tau$ to mean that expression $e$ has type $\tau$ under context $\Gamma$. The analysis first analyzes $e$ with post state $\{\mathcal{G}' \mid \Gamma'\}$, determines $e$'s type $\tau$ (second premise) and adds $v : \tau$ to $\Gamma'$. Our analysis relies on a type checker to determine the type of each expression.

$$\frac{\{\mathcal{G} \mid \Gamma\}\ e\ \{\mathcal{G}' \mid \Gamma'\}}{\{\mathcal{G} \mid \Gamma\}\ v \leftarrow e\ \{\mathcal{G}' \mid \Gamma'\}}\ \mathtt{asgn}$$

The assignment expression $v \leftarrow e$ simply gas-analyzes $e$.

$$\frac{\{\mathcal{G}_0 \mid \Gamma_0\}\ e\ \{\mathcal{G}_1 \mid \Gamma_1\} \qquad \{\mathcal{G}_1 \mid \Gamma_1\}\ e_1\ \{\mathcal{G}_2 \mid \Gamma_2\} \qquad \{\mathcal{G}_1 \mid \Gamma_1\}\ e_2\ \{\mathcal{G}_2 \mid \Gamma_2\}}{\{\mathcal{G}_0 \mid \Gamma_0\}\ \mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2\ \{\mathcal{G}_2 \mid \Gamma_2\}}\ \mathtt{if}$$

For `if` expressions, $e$ is analyzed under pre-state $\{\mathcal{G}_0 \mid \Gamma_0\}$ resulting in post-state $\{\mathcal{G}_1 \mid \Gamma_1\}$. This state is then copied to both branches $e_1$ and $e_2$, which both result in post-state $\{\mathcal{G}_2 \mid \Gamma_2\}$. We mandate that the post-gases $\mathcal{G}_2$ after both branches are equal, thus ensuring that both branches have equal gas cost. This is exactly where `Gas.deposit` operation is used to equalize the cost of both branches. Our tool automatically instruments the cheaper branch with `Gas.deposit`($n$) where $n$ is the difference in the post-gas of $e_1$ and $e_2$.

$$\frac{\{\mathcal{G}_0 \mid \Gamma_0\}\ e_1\ \{\mathcal{G}_1 \mid \Gamma_1\} \qquad \{\mathcal{G}_1 \mid \Gamma_1\}\ e_2\ \{\mathcal{G}_2 \mid \Gamma_2\}}{\{\mathcal{G}_0 \mid \Gamma_0\}\ e_1\ ;\ e_2\ \{\mathcal{G}_2 \mid \Gamma_2\}}\ \mathtt{compose}$$

Expression composition is standard; the intermediate state $\{\mathcal{G}_1 \mid \Gamma_1\}$ is the post-state for $e_1$ and the pre-state for $e_2$.

$$\frac{\{\mathcal{G} \mid \Gamma\}\ e\ \{\mathcal{G}' \mid \Gamma'\} \qquad \mathcal{G}' = 0}{\{\mathcal{G} \mid \Gamma\}\ \mathtt{return}\ e\ \{\mathcal{G}' \mid \Gamma'\}}\ \mathtt{ret}$$

We require that the post-gas of a return expression $\mathcal{G}' = 0$, thus ensuring the initial gas tank is completely used up for the function execution and the gas bound is exact. In case of branches, we require that the post-gas after each `return` expression is 0. The analysis rules for all other expressions are analogous and skipped for brevity.

### 3.3 Soundness of Analysis

We prove the soundness of the analysis by connecting the static gas analysis with the gas semantics. We define a program state $\sigma$ as a mapping from variables to their values. We formalize the gas semantics as $\sigma \vdash e \Downarrow^{\mu}_{\mu'} (v, \sigma')$ to define that the expression $e$ evaluates to value $v$ under program state $\sigma$ with resulting program state $\sigma'$. The annotations $\mu$ and $\mu'$ denote the gas tank value (as a natural number) before and after the evaluation of $e$.

We describe selected rules that impact the gas cost.

$$\frac{}{\sigma \vdash \texttt{tick}(n) \Downarrow^{\mu+n}_{\mu} ((), \sigma)}\ \texttt{TICK}$$

Executing $\texttt{tick}(n)$ consumes $n$ gas units from the tank. The value of `tick` is uninteresting and we use the convention that it evaluates to $()$.

$$\frac{}{\sigma \vdash \texttt{Gas.construct}(n) \Downarrow^{\mu+n}_{\mu} (n, \sigma)}\ \texttt{CONSTRUCT}$$

Semantically, we treat gas values as natural numbers. Thus, a variable of type $\texttt{Gas}(n)$ evaluates to $n$. The gas cost of constructing is $n$, so the difference in the initial and final gas tanks is $n$.

$$\frac{}{\{[v \mapsto n], \sigma\} \vdash \texttt{Gas.destruct}(v) \Downarrow^{\mu}_{\mu+n} ((), \sigma)}\ \texttt{DESTRUCT}$$

Destructing a variable with value $n$ (i.e., of type $\texttt{Gas}(n)$) adds $n$ to the gas tank. The value of destructing a gas variable is uninteresting and denoted by $()$. The variable is also removed from $\sigma$ since it is no longer available.

$$\frac{}{\sigma \vdash \texttt{Gas.deposit}(n) \Downarrow^{\mu+n}_{\mu} ((), \sigma)}\ \texttt{DEPOSIT}$$

Depositing gas into the user's account removes the same from the gas tank.

$$\frac{\begin{array}{c}\texttt{fn}\ [\mathcal{G}]\ \texttt{g}(x_1 : \tau_1, \ldots, x_n : \tau_n) \to \tau\ \{e\} \in \Sigma \\ \sigma_0 \vdash e_1 \Downarrow^{\mu_0}_{\mu_1} (v_1, \sigma_1) \qquad \ldots \qquad \sigma_{n-1} \vdash e_n \Downarrow^{\mu_{n-1}}_{\mu_n} (v_n, \sigma_n) \\ \sigma_n \vdash e[v_1, \ldots, v_n / x_1, \ldots, x_n] \Downarrow^{\mu_n}_{\mu'} (v, \sigma')\end{array}}{\sigma_0 \vdash \texttt{g}(e_1, \ldots, e_n) \Downarrow^{\mu_0}_{\mu'} (v, \sigma')}\ \texttt{CALL}$$

A function call to `g` evaluates each argument, then evaluates the body $e$ of `g` with the value of each argument $v_i$ substituted for the argument variable $x_i$. The body $e$ of `g` is looked up in the global signature $\Sigma$.

$$\frac{\sigma_0 \vdash e \Downarrow^{\mu_0}_{\mu_1} (v, \sigma_1)}{\sigma_0 \vdash \texttt{let}\ x = e \Downarrow^{\mu_0}_{\mu_1} ((), \{\sigma_1, [x \mapsto v]\})}\ \texttt{LET}$$

The `let` expression evaluates $e$ to $v$ with resulting state $\sigma_1$. It then assigns $v$ to $x$ and continues execution. The return value of the `let` expression is $()$. A similar rule holds for assignments. For `if` expressions, we consider two cases.

$$\frac{\sigma_0 \vdash e \Downarrow^{\mu_0}_{\mu_1} (\texttt{true}, \sigma_1) \qquad \sigma_1 \vdash e_1 \Downarrow^{\mu_1}_{\mu_2} (v, \sigma_2)}{\sigma_0 \vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \Downarrow^{\mu_0}_{\mu_2} (v, \sigma_2)}\ \texttt{TT}$$

$$\frac{\sigma_0 \vdash e \Downarrow^{\mu_0}_{\mu_1} (\texttt{false}, \sigma_1) \qquad \sigma_1 \vdash e_2 \Downarrow^{\mu_1}_{\mu_2} (v, \sigma_2)}{\sigma_0 \vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \Downarrow^{\mu_0}_{\mu_2} (v, \sigma_2)}\ \texttt{FF}$$

If $e$ evaluates to `true` with final tank $\mu_1$, we evaluate $e_1$ with initial tank $\mu_1$, otherwise we evaluate $e_2$ with tank $\mu_1$.

$$\frac{\sigma_0 \vdash e_1 \Downarrow_{\mu_1}^{\mu_0} (v_1, \sigma_1) \qquad \sigma_1 \vdash e_2 \Downarrow_{\mu_2}^{\mu_1} (v_2, \sigma_2)}{\sigma_0 \vdash e_1 \ ; \ e_2 \Downarrow_{\mu_2}^{\mu_0} (v_2, \sigma_2)} \ \texttt{COMPOSE}$$

Expression composition is standard; $\sigma_1$ and $\mu_1$ are the intermediate program state and tank value, respectively.

$$\frac{\sigma \vdash e \Downarrow_{\mu_1}^{\mu_0} (v, \sigma')}{\sigma \vdash \texttt{return}\ e \Downarrow_{\mu_1}^{\mu_0} (v, \sigma')} \ \texttt{RET}$$

Finally, `return` $e$ evaluates $e$. The semantics rules for the remaining expressions are analogous and skipped for brevity.

**Theorem 1 (Soundness).** *Given a function* $\texttt{fn}\ [\mathcal{G}]\ \texttt{g}(x_1 : \tau_1, \ldots, x_n : \tau_n)$ *and a program state* $\sigma$*, if* $\sigma \vdash \texttt{g}(v_1, \ldots, v_n) \Downarrow_{\mu'}^{\mu} (v, \sigma')$*, then* $\mu - \mu' = \mathcal{G}$*.*

Intuitively, the gas soundness theorem states that if a function call to `g` executes under program state $\sigma$ with initial tank $\mu$ and final tank $\mu'$, the difference $\mu - \mu'$ is exactly equal to the gas bound $\mathcal{G}$. Thus, the static gas analysis provides an exact bound on the gas cost at runtime. The theorem is proved by induction on the gas semantics judgment.

## 4 Implementation and Evaluation

We have implemented a prototype for GasBoX in OCaml (1866 lines of code). The prototype contains a lexer and parser (321 lines), tick instrumentation engine (188 lines), pretty printer (207 lines), an arithmetic solver (309 lines) and gas analyzer (841 lines). The lexer and parser are implemented in Menhir [35], an LR(1) parser generator for OCaml.

***Tick Instrumentation.*** Once the program has been parsed and represented as an abstract syntax tree, we insert the `tick` expressions following Section 3.1. Since the tick amounts for `pack`, `unpack` and `move` depend on the size of the type being operated, we precompute the size of all types in the program. The instrumentation engine takes the sizes and the cost model (values of $\mathcal{C}_{\texttt{i}}$'s) as input and inserts the tick expressions. Programmers are free to specify their own cost model and the analysis computes the bound w.r.t. specified cost model.

***Gas Analysis.*** The gas analyzer iterates through the function declarations, taking the initial gas bound and definition as input, and verifying whether the bound is exact. To enhance usability, we have designed our gas analyzer with a specific focus towards the quality of error messages. To this end, the parser stores the *extent* (source code location) information in the abstract syntax tree. If the gas tank runs below 0 at any program location, the program is pretty printed back to the user with the source location highlighted.

### 4.1 Evaluation

We evaluate GasBoX by implementing standard smart contracts in our language, and verifying their gas bounds. We highlight some interesting examples, particularly the ones that involve amortization to handle unbounded computation. All our experiments use the cost model assigning $\mathcal{C}_{\texttt{i}} = 1$ for all $i$.

***Paying Interest on Bank Accounts.*** We implement a standard bank account contract, which provides the services of signing up to create an account, withdrawing and depositing money, and checking balance. The bank provides an additional facility of paying interest to each account holder periodically. The bank stores gas inside accounts to pay for the gas cost of paying interest.

```
resource GasBalance {
  balance : Coin,
  gas : Gas(65)      // utilized to pay interest periodically
}
resource Bank {
  nogas_accounts : Map<address, Coin>,
  gas_accounts : Map<address, GasBalance>
}
fn [201] recharge(bank : &Bank)
fn [29] payInterest(bank : &Bank)
fn [34] signup(bank : &Bank, amount : Coin)
fn [122] balance(bank : &Bank) -> int
fn [148] deposit(bank : &Bank, amount : Coin)
fn [187] withdraw(bank : &Bank, amount : int) -> Coin
```

The contract defines the resource type `GasBalance` for accounts containing gas. For our cost model, we need 65 gas units in each account for paying interest. The `Bank` type contains two maps: `gas_accounts` and `nogas_accounts` for accounts with and without gas respectively indexed by the address of the account holder. The contract provides a `recharge` function that replenishes gas in the sender's account, effectively removing it from `nogas_accounts` and adding it to `gas_accounts`. The `payInterest` function recursively removes an account from `gas_accounts`, consumes the gas stored in it to pay the interest, and adds it to `nogas_accounts`. Thus, it is the account holder's responsibility to periodically replenish the gas in their account by issuing the `recharge` function; the `payInterest` function only pays interest to accounts stored in `gas_accounts`. In addition, the contract provides the standard `signup`, `balance`, `deposit` and `withdraw` functions to create an account, check balance, deposit and withdraw money, respectively. The exact gas bound for each function is shown in square brackets [·] along with the declaration.

The gas amortization provides the following benefits: (i) mitigating denial-of-service attacks since the gas bound of `payInterest` no longer depends on the number of bank accounts, (ii) equitable gas distribution since each account holder is responsible for covering the gas cost of paying interest on their account.

***Voting.*** We implement a simple voting contract that provides two functions: a `vote` function to allow voters to cast their vote and a `count` function that counts the votes and computes the winner. The contract amortizes the cost of counting votes by storing gas inside the votes cast.

```
resource Votes {
  num_votes : int,
  gas : Gas(69) }      // utilized to count votes when election ends
```

```
fn [114] vote(elec : &Map<address, Votes>, candidate : address)
fn [55] count(elec : &Map<address, Votes>) -> address
```

The contract defines the resource type `Votes` used to store the votes for a particular candidate. The type contains two fields: `num_votes` denotes the number of votes for the candidate, and `gas` stores 69 gas units to pay for counting votes later. The `vote` function takes two arguments: `elec` contains the map storing the votes indexed by the address of the candidate, and `candidate` is the address of the candidate the sender wants to vote for. The function increments the number of votes in `candidate`'s name by 1. The `count` function takes `elec` as argument, iterates over the map, and consumes the gas stored inside it to compute the winner of the election. The exact gas bound for both functions is a constant and described alongside the declaration. This contract also provides the advantages of mitigating denial-of-service attacks and equitable gas distribution.

***Other Contracts.*** We have implemented a total of 13 contracts in our language, and verified their gas bound with GasBoX. We briefly describe each contract.

1. **auction**: unamortized version of auction providing support for users to *pull* their bids out of the contract.
2. **bank**: naïve bank account with no functionality to pay interest.
3. **ERC 20**: technical standard for token implementation on Ethereum defining a list of rules Ethereum tokens should follow [1].
4. **escrow**: contract to exchange bonds between two parties.
5. **insurance**: contract processing flight delay insurance claims after verifying them with a trusted third party.
6. **voting**: election contract described earlier in this section.
7. **wallet**: standard contract allowing users to store money on the blockchain.
8. **ethereumpot**: standard lottery contract on Ethereum.
9. **puzzle**: contract rewarding users who solve a computational puzzle and submit the solution.
10. **amort. auction**: amortized auction described in Section 2.
11. **amort. bank**: amortized bank account paying interest periodically as described earlier in this section.
12. **tether**: stable coin contract allowing exchange of digital tokens pegged to fiat currencies e.g. dollars, euros, etc. [2].
13. **libra system**: standard library contract with recursive functions for configuration of third-party validators

Contracts 1-7 have been borrowed from the Nomos project [17], ethereumpot from the Gastap project [6], puzzle from the Oyente project [32], tether from the Tether ERC 20 token contract [2] and libra system from the Libra blockchain [8] and reimplemented in our language.

Table 1 compiles the results of evaluating GasBoX on the implemented contracts. For each contract, we present the lines of code (LOC), number of resource and struct definitions (Defs), number of function definitions (Funcs), whether the functions are recursive and require amortization, and the gas analysis time in

| No. | Contract Name | LOC | Defs | Funcs | Rec? | Time (µs) |
|---|---|---|---|---|---|---|
| 1 | auction | 52 | 4 | 3 | No | 60.08 |
| 2 | bank | 146 | 6 | 9 | No | 187.15 |
| 3 | ERC 20 | 107 | 4 | 8 | No | 159.97 |
| 4 | escrow | 140 | 3 | 4 | No | 152.82 |
| 5 | insurance | 43 | 3 | 2 | No | 32.90 |
| 6 | voting | 82 | 3 | 5 | Yes | 116.10 |
| 7 | wallet | 74 | 4 | 4 | No | 128.03 |
| 8 | ethereumpot | 259 | 4 | 9 | No | 1532.07 |
| 9 | puzzle | 62 | 2 | 4 | No | 61.03 |
| 10 | amort. auction | 70 | 6 | 5 | Yes | 75.10 |
| 11 | amort. bank | 189 | 9 | 10 | Yes | 254.15 |
| 12 | tether | 382 | 9 | 20 | No | 2577.06 |
| 13 | libra system | 123 | 5 | 7 | Yes | 126.12 |
| | **Total** | **1729** | **62** | **90** | | **5462.58** |

**Table 1.** Evaluation of GasBoX. LOC = lines of code; Defs = #type definitions; Funcs = #function definitions; Rec? = recursive functions in the contract?; Time (µs) = gas analysis time in microseconds.

microseconds. The experiments were run on an Intel Core i5 1.6 GHz dual-core processor with 16 GB DDR3L memory.

The evaluation demonstrates that the analysis is highly efficient with an overhead of less than 1 millisecond for all but two contracts. This indicates that GasBoX can be effectively utilized by miners to determine the exact gas bound. Moreover, this overhead is offset by the elimination of dynamic gas metering from the virtual machine. The error messages were precise and helpful in guiding us provide the correct initial gas bounds. Since the `Gas.deposit` operations were automatically inserted, we could remain oblivious of the exact cost model and difference in gas costs of different branches.

## 5 Related Work

Traditionally, resource analysis is grounded in deriving and solving recurrence relations, an approach introduced to analyze simple Lisp programs [41]. Since then, it has been applied to both imperative [3, 21, 7] and functional programs [9, 15]. Amortization [39] was first integrated with resource analysis to automatically analyze heap usage of first-order functional programs [29]. In the context of functional languages, this technique has been applied to derive polynomial [28] and multivariate bounds [26] for first-order and higher-order programs [27] as well as programs with lazy evaluation [37]. For imperative programs, amortization has been utilized to derive bounds based on lexicographic ranking functions [38] and intervals [13], and has been extended to analyze object-oriented programs [30]. In contrast to the above works that focus on upper bounds, GasBoX verifies exact bounds for programs and is applicable to smart contracts.

Security analysis and safety verification of smart contracts have been extensively studied in prior work [24, 40, 31, 10, 32]. MadMax [23] automatically detects gas-focused vulnerabilities with high confidence. The analysis is based on a decompiler that extracts control and data flow information from EVM bytecode, and a logic-based analysis specification that produces a high-level program model. GASPER [14] is an analysis tool for EVM bytecode that relies on symbolic execution and the Z3 SMT solver [34] to identify 7 gas-costly programming patterns such as dead code, expensive and repeated computations in a loop, etc. GasBoX differs from these works by verifying gas cost, instead of identifying vulnerabilities related to gas.

Most closely related to GasBoX are languages and analysis tools for estimating upper gas bounds on contracts. Scilla [36] is an intermediate-level language which disallows loops and general recursion and infers gas usage of a function as a polynomial of the size of its parameters and contract fields in linear time. In contrast, GasBoX allows recursion and bounds are proven sound w.r.t. a gas semantics. Nomos is a programming language [17] based on resource-aware session types [19, 18] that utilizes LP (linear programming) solving to automatically derive upper gas bounds on implemented contracts. Gastap [6] infers gas bounds on contracts implemented in Solidity [16] or EVM bytecode in terms of size of the input parameters, contract state and gas consumption. The inference procedure requires construction of control-flow graphs, decompilation to a high-level representation, inferring size relations, generating and solving gas equations. Gasol [5] is an extension to Gastap which offers a variety of cost models to measure the cost of, for e.g., only storage opcodes, selected family of gas-consumption opcodes, selected program line, etc. It further detects under-optimized storage patterns and automatic optimization of such patterns. Marescotti et. al. [33] employ symbolic model checking to modularly enumerate all gas consumption paths based on unwinding loops to a limit. For each path, it then computes the environment state to force that path and simulates the transaction under the state to obtain an exact worst-case gas bound. GasBoX differs from these tools in its goal of providing miners with a trusted exact gas bound which can be verified in linear time and eliminating dynamic gas metering.

## 6 Conclusion

This paper presented a Hoare-logic style gas-analysis framework for smart contracts. This framework verifies exact gas bounds in linear-time and relies on amortization to handle unbounded computation. The verified gas bounds are proven sound w.r.t. a gas semantics. The framework has been implemented as a tool called GasBoX in the context of a simplistic programming language. GasBoX has been evaluated on several standard smart contracts demonstrating its efficiency and expressivity.

In the future, we plan to use more sophisticated underlying logics such as SMT solvers, carefully weighing the balance of expressivity and efficiency of the gas-analysis framework. We would also like to handle copying of unbounded data structures such as maps. We also plan to add automatic inference of gas bounds by generating linear equations and solving them using efficient off-the-shelf LP solvers. Finally, our approach is largely independent of the target language, and we would like to extend our analysis tool to languages such as Solidity and Move.

## References

1. Erc20 token standard. https://theethereum.wiki/w/index.php/ERC20_Token_Standard (december 2018), accessed: 2018-02-027
2. Tether: Digital money for a digital age. https://tether.to/ (Apr 2020), accessed: 2020-04-29
3. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Cost analysis of java bytecode. In: De Nicola, R. (ed.) Programming Languages and Systems. pp. 157–172. Springer Berlin Heidelberg, Berlin, Heidelberg (2007)
4. Albert, E., Arenas, P., Genaim, S., Herraiz, I., Puebla, G.: Comparing cost functions in resource analysis. In: van Eekelen, M., Shkaravska, O. (eds.) Foundational and Practical Aspects of Resource Analysis. pp. 1–17. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
5. Albert, E., Correas, J., Gordillo, P., Román-Díez, G., Rubio, A.: Gasol: Gas analysis and optimization for ethereum smart contracts (2019)
6. Albert, E., Gordillo, P., Rubio, A., Sergey, I.: Running on fumes. In: Ganty, P., Kaâniche, M. (eds.) Verification and Evaluation of Computer and Communication Systems. pp. 63–78. Springer International Publishing, Cham (2019)
7. Alonso-Blas, D.E., Genaim, S.: On the limits of the classical approach to cost analysis. In: Miné, A., Schmidt, D. (eds.) Static Analysis. pp. 405–421. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
8. Baudet, M., Ching, A., Chursin, A., Danezis, G., Garillot, F., Li, Z., Malkhi, D., Naor, O., Perelman, D., Sonnino, A.: State machine replication in the libra blockchain (2019), https://developers.libra.org/docs/assets/papers/libra-consensus-state-machine-replication-in-the-libra-blockchain.pdf
9. Benzinger, R.: Automated higher-order complexity analysis. Theoretical Computer Science **318**(1), 79 – 103 (2004). https://doi.org/https://doi.org/10.1016/j.tcs.2003.10.022, http://www.sciencedirect.com/science/article/pii/S0304397503005279, implicit Computational Complexity
10. Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Gollamudi, A., Gonthier, G., Kobeissi, N., Kulatova, N., Rastogi, A., Sibut-Pinote, T., Swamy, N., Zanella-Béguelin, S.: Formal verification of smart contracts: Short paper. In: Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security. pp. 91–96. PLAS '16, ACM, New York, NY, USA (2016). https://doi.org/10.1145/2993600.2993611, http://doi.acm.org/10.1145/2993600.2993611
11. Blackshear, S., Cheng, E., Dill, D.L., Gao, V., Maurer, B., Nowacki, T., Pott, A., Qadeer, S., Rain, D.R., Sezer, S., et al.: Move: A language with programmable resources (2019)
12. Carbonneaux, Q., Hoffmann, J., Reps, T., Shao, Z.: Automated resource analysis with coq proof objects. In: Majumdar, R., Kunčak, V. (eds.) Computer Aided Verification. pp. 64–85. Springer International Publishing, Cham (2017)
13. Carbonneaux, Q., Hoffmann, J., Shao, Z.: Compositional certified resource bounds. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 467–478. PLDI '15, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2737924.2737955, https://doi.org/10.1145/2737924.2737955
14. Chen, T., Li, X., Luo, X., Zhang, X.: Under-optimized smart contracts devour your money. In: 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER). pp. 442–446 (2017)

15. Danielsson, N.A.: Lightweight semiformal time complexity analysis for purely functional data structures. In: Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 133–144. POPL '08, Association for Computing Machinery, New York, NY, USA (2008). https://doi.org/10.1145/1328438.1328457, `https://doi.org/10.1145/1328438.1328457`
16. Dannen, C.: Introducing Ethereum and Solidity: Foundations of Cryptocurrency and Blockchain Programming for Beginners. Apress, USA, 1st edn. (2017)
17. Das, A., Balzer, S., Hoffmann, J., Pfenning, F.: Resource-aware session types for digital contracts. CoRR **abs/1902.06056** (2019), `http://arxiv.org/abs/1902.06056`
18. Das, A., Hoffmann, J., Pfenning, F.: Parallel complexity analysis with temporal session types. Proc. ACM Program. Lang. **2**(ICFP) (Jul 2018). https://doi.org/10.1145/3236786, `https://doi.org/10.1145/3236786`
19. Das, A., Hoffmann, J., Pfenning, F.: Work analysis with resource-aware session types. In: Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science. pp. 305–314. LICS '18, ACM, New York, NY, USA (2018). https://doi.org/10.1145/3209108.3209146, `http://doi.acm.org/10.1145/3209108.3209146`
20. Fischer, M.J., Rabin, M.O.: Super-exponential complexity of presburger arithmetic. In: Caviness, B.F., Johnson, J.R. (eds.) Quantifier Elimination and Cylindrical Algebraic Decomposition. pp. 122–135. Springer Vienna, Vienna (1998)
21. Flores-Montoya, A., Hähnle, R.: Resource analysis of complex programs with cost equations. In: Garrigue, J. (ed.) Programming Languages and Systems. pp. 275–295. Springer International Publishing, Cham (2014)
22. Girard, J.Y.: Linear logic. Theoretical Computer Science **50**(1), 1 – 101 (1987). https://doi.org/https://doi.org/10.1016/0304-3975(87)90045-4, `http://www.sciencedirect.com/science/article/pii/0304397587900454`
23. Grech, N., Kong, M., Jurisevic, A., Brent, L., Scholz, B., Smaragdakis, Y.: Madmax: Surviving out-of-gas conditions in ethereum smart contracts. Proc. ACM Program. Lang. **2**(OOPSLA), 116:1–116:27 (Oct 2018). https://doi.org/10.1145/3276486, `http://doi.acm.org/10.1145/3276486`
24. Grishchenko, I., Maffei, M., Schneidewind, C.: Foundations and tools for the static analysis of ethereum smart contracts. In: Chockler, H., Weissenbacher, G. (eds.) Computer Aided Verification. pp. 51–78. Springer International Publishing, Cham (2018)
25. Gulwani, S.: Speed: Symbolic complexity bound analysis. In: Bouajjani, A., Maler, O. (eds.) Computer Aided Verification. pp. 51–62. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)
26. Hoffmann, J., Aehlig, K., Hofmann, M.: Multivariate amortized resource analysis. ACM Trans. Program. Lang. Syst. **34**(3) (Nov 2012). https://doi.org/10.1145/2362389.2362393, `https://doi.org/10.1145/2362389.2362393`
27. Hoffmann, J., Das, A., Weng, S.C.: Towards automatic resource bound analysis for ocaml. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages. p. 359–373. POPL 2017, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3009837.3009842, `https://doi.org/10.1145/3009837.3009842`
28. Hoffmann, J., Hofmann, M.: Amortized resource analysis with polynomial potential. In: Gordon, A.D. (ed.) Programming Languages and Systems. pp. 287–306. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)

29. Hofmann, M., Jost, S.: Static prediction of heap space usage for first-order functional programs. In: Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 185–197. POPL '03, ACM, New York, NY, USA (2003). https://doi.org/10.1145/604131.604148, `http://doi.acm.org/10.1145/604131.604148`
30. Hofmann, M., Jost, S.: Type-based amortised heap-space analysis. In: Proceedings of the 15th European Conference on Programming Languages and Systems. p. 22–37. ESOP'06, Springer-Verlag, Berlin, Heidelberg (2006)
31. Lahiri, S.K., Chen, S., Wang, Y., Dillig, I.: Formal specification and verification of smart contracts for azure blockchain. CoRR **abs/1812.08829** (2018), `http://arxiv.org/abs/1812.08829`
32. Luu, L., Chu, D.H., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. pp. 254–269. CCS '16, ACM, New York, NY, USA (2016). https://doi.org/10.1145/2976749.2978309, `http://doi.acm.org/10.1145/2976749.2978309`
33. Marescotti, M., Blicha, M., Hyvärinen, A.E.J., Asadi, S., Sharygina, N.: Computing exact worst-case gas consumption for smart contracts. In: International Symposium on Leveraging Applications of Formal Methods ISoLA 2018: Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice. Springer, Springer, Cyprus (2018), `https://link.springer.com/chapter/10.1007\%2F978-3-030-03427-6_33`
34. de Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
35. Pottier, F., Régis-Gianas, Y.: Menhir Reference Manual (2019)
36. Sergey, I., Nagaraj, V., Johannsen, J., Kumar, A., Trunov, A., Hao, K.C.G.: Safer smart contract programming with scilla. Proc. ACM Program. Lang. **3**(OOPSLA) (Oct 2019). https://doi.org/10.1145/3360611, `https://doi.org/10.1145/3360611`
37. Simões, H., Vasconcelos, P., Florido, M., Jost, S., Hammond, K.: Automatic amortised analysis of dynamic memory allocation for lazy functional programs. In: Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming. p. 165–176. ICFP '12, Association for Computing Machinery, New York, NY, USA (2012). https://doi.org/10.1145/2364527.2364575, `https://doi.org/10.1145/2364527.2364575`
38. Sinn, M., Zuleger, F., Veith, H.: A simple and scalable static analysis for bound analysis and amortized complexity analysis. In: Biere, A., Bloem, R. (eds.) Computer Aided Verification. pp. 745–761. Springer International Publishing, Cham (2014)
39. Tarjan, R.: Amortized computational complexity. SIAM Journal on Algebraic and Discrete Methods **6**(2), 306–318 (1985)
40. Tikhomirov, S., Voskresenskaya, E., Ivanitskiy, I., Takhaviev, R., Marchenko, E., Alexandrov, Y.: Smartcheck: Static analysis of ethereum smart contracts. In: 2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB). pp. 9–16 (May 2018)
41. Wegbreit, B.: Mechanical program analysis. Commun. ACM **18**(9), 528–539 (Sep 1975). https://doi.org/10.1145/361002.361016, `https://doi.org/10.1145/361002.361016`

42. Wood, G.: Ethereum: A secure decentralised generalised transaction ledger eip-150 revision (759dccd - 2017-08-07) (2017), `https://ethereum.github.io/yellowpaper/paper.pdf`, accessed: 2018-01-03