Efficient static analysis of Marlowe contracts

Pablo Lamela Seijas[1 [0000−0002−1730−1219]], David Smith[1 [0000−0003−1859−8007]], and Simon Thompson[1,2 [0000−0002−2350−301X]]

[1] IOHK, Hong Kong, pablo.lamela@iohk.io, david.smith@tweag.io, simon.thompson@iohk.io
[2] School of Computing, University of Kent, UK, s.j.thompson@kent.ac.uk

Abstract. SMT solvers can verify properties automatically and efficiently, and they offer increasing flexibility on the ways those properties can be described. But it is hard to predict how those ways of describing the properties affect the computational cost of verifying them.
In this paper, we discuss what we learned while implementing and optimising the static analysis for Marlowe, a domain specific language for self-enforcing financial smart-contracts that can be deployed on a blockchain.

1 Introduction

Thanks to static analysis, we can automatically check beforehand whether any payments promised by a Marlowe [12,13] contract can be fulfilled in every possible execution of the contract. If a Marlowe contract has passed the static analysis, we will have a very high assurance that whenever the contract says it will make a payment, the contract will indeed have enough money available.

Marlowe's static analysis relies on SMT solvers, which are able to check efficiently whether a set of constraints is satisfiable. The main property for us is whether a contract will have enough money for all payments to be made in full.

Thanks to state of the art libraries like SBV [7], we can describe those constraints in a high level language. In the case of SBV, we can write properties as Haskell functions, with few restrictions on how those functions are implemented; SBV automatically translates those functions to SMTLib format [3], a language many SMT solvers understand. However, high level abstractions often have a tradeoff with efficiency, and static analysis can become very expensive computationally if implemented naïvely, because it is NP-complete in the general case.

This paper contributes a number of approaches that can be used when optimising static analysis, with examples extracted from a case-study where we applied these approaches. Our approach aims to ensure correctness of the optimisations by combining the use of property-based testing and with verification in an automated theorem prover to establish properties of the optimisations. We also present empirical data that measures the effect of some of the optimisations on the implementation of our Marlowe case-study.

The techniques described helped us reduce the analysis time of an implementation that followed the semantics closely and took a couple of minutes to analyse contracts of a few kilobytes, to one where the same contract takes less than a

second to analyse, and where a four-person crowdfunding contract that fully expanded occupies about 19 megabytes, can be analysed in around 10 minutes. We have classified optimisation techniques as lightweight and heavyweight.

Lightweight modifications are local and can be done without fundamentally changing the implementation. We consider three main ideas:

- **Removing unnecessary parts from the analysis**. If they do not affect the property that is being verified then we can just remove them.
- **Avoiding high level abstractions**. High level abstractions aid reasoning and avoid errors, but also introduce complexity that may not be necessary.
- **Reducing the search space by normalising parameters**. If there are several ways of representing some inputs, and the different representations have no impact on the analysis, we can remove all but one for analysis.

Heavyweight modifications are more fundamental approaches that require considerable changes to the structure of the implementation. But these optimisations can also translate in important reductions to execution time and memory usage, as shown by the experiments we report in Section 6. We consider two main ideas:

- **Reducing the search space by using normalised execution paths relevant to the property**. Instead of using the search space to model inputs, we use it to model possible executions, and we only represent each equivalence class of executions once, i.e: we represent them in a normal form.
- **Minimizing the representation of inputs and outputs**. We encode inputs and outputs as concisely as possible, discarding inferable information.

In the following sections, we introduce the semantics of Marlowe as a case study (Section 2), and we cover a general approach to static analysis (Section 3). We then, in Section 4 explore in Section the lightweight and heavyweight optimisation techniques in more detail, and illustrate them with examples of how they apply to Marlowe static analysis. In Section 5, we illustrate the use of property based testing on the implementation with heavyweight optimisations. Finally, we present empirical results that show the effect of heavyweight optimisations on the execution time and memory usage of Marlowe's static analysis (Section 6).

2 Marlowe design and semantics

Firstly, we introduce the semantics of Marlowe, the guarantees that are offered implicitly by the semantics, and how the design choices facilitate static analysis and make it decidable. A more detailed explanation can be found in [12].

2.1 Structure of Marlowe contracts

The Marlowe language is realised as a set of mutually recursive Haskell data types. Marlowe contracts regulate interactions between a finite number of participants, determined before the contract is deployed.

Marlowe contracts are able to receive payments, store money and tokens, ask for input from its participants, and redistribute stored money and tokens among participants. A contract determines when and which of these actions may be carried out. Participants may correspond to either individual public keys or to tokens (`Role`s). In turn, `Role`s may be controlled by other contracts.

The main data type in Marlowe is called `Contract`, and it represents the logic and actions that the contract allows or enforces. The outmost constructs of the `Contract` represent the actions that will be enforced first and, as those constructs become enforced, the `Contract` will evolve into one of its continuations (subcontracts), and the process will continue until only the construct `Close` remains.

There are 5 constructs of type `Contract`:

- `Close` – signals the end of the life of a contract. Once it is reached, all the money and tokens stored in the accounts of the contract will be refunded to the owner of each of the respective accounts.
- `If` – immediately decides how the contract must continue from between two possibilities, depending on a given Boolean condition that we call an `Observation`. An `Observation` may depend on previous choices by participants, on amounts of money and tokens in the contract, or other factors.
- `Let` – immediately stores a `Value` for later use. Expressions of type `Value` in Marlowe are evaluated to integer values and they depend on information available to the contract at the time of evaluation. For example, a `Let` construct record the amount of money in an account at a point in time.
- `When` – waits for an external input. The `When` construct also specifies a timeout slot[3]: after this slot has been reached, the `When` construct expires, and no longer accepts any input. There are three kinds of input:
  - `Deposit` – Waits for a participant to deposit an amount of money or tokens (specified as a `Value`) in an account of the contract.
  - `Choice` – Waits for a participant to make a choice. A choice is represented as an integer number from a set specified by the contract.
  - `Notify` – Waits for an `Observation` to be true. Because contracts are reactive (they cannot initiate transactions), it is necessary for an external actor (not necessarily a participant) to `Notify` the contract.
- `Pay` – immediately makes a payment between accounts of the contract, or from an account of the contract to a given participant. The amount transferred is specified as a `Value`.

2.2 Semantics

As we mentioned in the previous section, Marlowe contracts are passive: their code is executed as part of the validation of transactions that are submitted to the blockchain. Transactions need to be submitted by participants or their representatives (e.g. user wallets) and validation is atomic and deterministic.

Each transaction may include a list of inputs, a set of signatures, a slot interval, a set of input UTxOs (incoming money and tokens), and a set of outputs

[3]Slots are blockchain's proxy for time, they are added to the blockchain periodically.

(outgoing money and tokens). We use a slot interval because it is very difficult to know the exact slot in which the transaction will be included in practice. For a transaction to be valid in Marlowe, the transaction must have the same effect for every slot within that slot interval (be deterministic). For example, if a transaction has a minimum slot number lower than timeout, and a maximum slot that is greater, then the transaction will fail with the error `AmbiguousSlotInterval`.

A key aspect of the Marlowe semantics is that it checks that a particular transaction is valid given the current state and contract. Because transactions are deterministic, there should be no reason why someone accidentally sends a transaction that is invalid for a given `State` and `Contract`, since it will only result in a cost to that participant. However, it is still possible that, due to a race condition, a participant will send a transaction that no longer applies to a running `Contract` and `State`, but such a transaction would simply be ignored by the blockchain.

The type signature of the transaction validation function is:

```
computeTransaction :: TransactionInput -> State -> Contract
                   -> TransactionOutput
```

This function can be factored into four main functions:

- `reduceContractStep` - this function executes the topmost construct that does not require an input to be executed (i.e: anything but a `When` that has not expired or a `Close` when accounts are empty). It only simplifies the `When` construct if it has expired (i.e. the timeout specified in the `When` is less than or equal to the minimum slot number). In the case of the `Close` contract, it only refunds one of the accounts at each invocation.
- `reduceContractUntilQuiescent` - this function calls `reduceContractStep` repeatedly until it has no further effect,
- `applyInput` - this function processes one single input. The topmost construct must be a `When` that is expecting that particular input and has not expired.
- `applyAllInputs` - this function processes a list of inputs. It calls `applyInput` for each of the inputs in the list, and calls `reduceContractUntilQuiescent` before and after every call to `applyInput`.

The `State` stores information about the amount of money and tokens in the contract at a given time, together with the choices made, `Let` bindings made, and a lower bound for the current slot:

```
data State = State { accounts    :: Map AccountId Money
                   , choices     :: Map ChoiceId ChosenNum
                   , boundValues :: Map ValueId Integer
                   , minSlot     :: Slot }
```

2.3 Extra considerations

Many of the design decisions behind Marlowe have been made with the aim of preventing potential errors. For example:

- **Classification of money and tokens into accounts** separates concerns. A Marlowe contract will never spend more money or tokens than there are in an account, even if there is more available in the contract. But a payment for more than there is will not fail, it will pay as much as is available, in order to remain as close as possible to the original intention of the contract.
- **Account identifiers include an account owner**. An account owner is a participant that will get the money or tokens remaining in an account when a contract terminates. At the same time, the only construct that can pause the execution of a contract is the `When` construct, which has a timeout, this ensures that all contracts eventually expire and terminate. Together, these properties ensure that no money or tokens are locked in the contract forever.
- **No negative deposits or payments**. Marlowe treats negative amounts in deposits and payments as zero. At the same time, if there happens to be a request for a deposit with a negative amount, the contract will still wait for a null deposit to be made, and it will continue as if everything is correct. This way execution of the contract is disturbed as little as possible.
- **Upper limit in the number of inputs** that a Marlowe contract can accept throughout its lifetime. This limit is implied by the path with maximum number of nested `When` constructs in the contract, since only one input per `When` can be accepted. At the same time, transactions with no effect on the contract are invalid, thus there is a limit on the maximum number of transactions a contract can accept throughout its life too. This bound prevents DoS attacks, and it makes static analysis easier. We discuss further in Section 3.

3 Making Marlowe semantics symbolic

In this section, we briefly present and reflect on a technique that can be used to convert a concrete implementation of a Haskell function into a symbolic one by using the SBV library [7], and to use this symbolic implementation for static analysis. In particular, we explore this technique in the context of the Marlowe.

This approach corresponds to our first attempt at implementing static analysis for Marlowe contracts, and it is a systematic approach that can be carried out with very few assumptions.

3.1 Overview

The SBV library supports implementing Haskell functions in a way that the same implementation can be used:

- With concrete parameters, as a normal Haskell function.
- With symbolic parameters, so that properties can be checked for satisfiability using an SMT solver.
- As part of QuickCheck properties, for random testing.

Parameters that can be used symbolically are wrapped in a monad called `SBV`. Values that depend on symbolic values must also be wrapped in the `SBV` monad.

Our semantics transaction processing function would thus become:

```
computeTransaction :: SBV TransactionInput -> SBV State
                   -> SBV Contract -> SBV TransactionOutput
```

We just need a function `playTrace` that takes a list of transactions and calls `computeTransaction` for each. We can then write our property to state that the output of `playTrace` does not have any failed payments (or other warnings). We can ask SBV to find an input transaction list that breaks the property. However, there are a couple of issues with this approach, we review them in Section 3.2.

3.2 Additional considerations

At the time of writing, SBV does not fully support complex custom data types, but it provides symbolic versions for `Either` and `Tuple` types. Our original implementation of Marlowe's static analysis overcomes this limitation by generating conversion functions using using Template Haskell [16]. This allows static analysis to remain similar to the semantics. For example, the following data structure:

```
data Input = IDeposit AccountId Party Money
           | IChoice ChoiceId ChosenNum
           | INotify
```

Would be translated to the following type synonym:

```
type SInput = SBV (Either (AccountId, Party, Money)
                          (Either (ChoiceId, ChosenNum)
                                  ()))
```

But this approach cannot address recursive datatypes, let alone mutually recursive datatypes. And the `Contract` definition uses mutual recursion.

Even if we could use symbolic corecursive datatypes, SMT solvers have another general limitation: if termination of a function is not bounded by a concrete value, SMT solvers may not terminate when determining the satisfiability of a property about the function. We discuss how to address this in Section 3.3.

3.3 Adapting the semantics

In order to guarantee termination of the analysis, we need a concrete bound. Related work often addresses this problem by manually establishing an artificial bound on the amount of computation, e.g: limiting the number of computation steps analysed, the number of times loops are unrolled [5,8,9].

Marlowe has natural bounds, given a concrete contract, we can infer:

- The maximum number of inputs that can have an effect on the contract
- The maximum number of transactions that can have an effect on the contract
- All of the account, choice, and `Let` identifiers that will be used in the contract
- The number of participants that will participate in the contract

From this data, we can also deduce an upper bound for:

- The number of times that `computeTransaction`, `reduceContractStep`, and `applyInput` may be called.
- The number of accounts that the contract will use, and the number of elements there may be in each of the associative maps that comprise the `State` of the contract at any given point of the execution.

Marlowe `Contract`s are finite, and every call to `reduceContractStep` will either make no progress or remove one of the constructs, with the exception of `Close`. In the case of `Close`, every call to `reduceContractStep` refunds one account, and the number of accounts is also bounded, since each needs to be mentioned in the `Contract`. Thus, the symbolic transaction processing function becomes:

```
computeTransaction :: SBV TransactionInput -> SBV State
                   -> Contract -> SBV TransactionOutput
```

There is one more problem: the output `Contract` returned by the function, which is wrapped inside the `TransactionOutput`, is symbolic, since it depends on the current `TransactionInput` and `State`, which are both symbolic.

We get around this by using a continuation style. Instead of returning the `TransactionOutput`, we take a continuation function that takes the concrete `Contract` and the symbolic version of `TransactionOutput` without the `Contract`.

Thus, the symbolic transaction processing function will look something like:

```
computeTransaction :: SymVal a => SBV TransactionInput
                   -> SBV State -> Contract
                   -> (SBV TransactionOutput -> Contract -> SBV a)
                   -> SBV a
```

In practice, we also include some extra information about bounds, and we make some other parts of `TransactionOutput` concrete.

4 Making static analysis more efficient

In this section, we explain the optimisation techniques in more detail, and we illustrate them with examples of their application to Marlowe's static analysis.

4.1 Lightweight modifications

Lightweight modifications are local, which means it is less likely that we will introduce reasoning errors when implementing them.

Removing unnecessary parts from the analysis. When we use the same or similar code for both the analysis and the implementation, we may end up including code that is not relevant to the analysis.

In the case of Marlowe, this was the case of the `Close` construct. The `Close` construct refunds all the money and tokens remaining in the accounts. The

number of times that `reduceContractStep` needs to be called depends on how many accounts have money left, and because this information is symbolic, there are many potential ways in which execution can unfold. All these paths need to be represented as constraints, which makes analysing `Close` very costly.

Fortunately, as it turns out, we do not need to analyse `Close` at all. Because it is impossible for the `Close` construct to produce a failed payment or any other warning (we have proven this[4] using Isabelle [15]). `Close` only pays as much as available, so we can safely remove it from the analysis.

Avoiding high level abstractions. High level libraries like SBV, and even standards like SMTLib, support the use and construction of high level abstractions, e.g: custom data-types, list, sets... Unfortunately, even though high level abstractions aid reasoning about code, they often prevent optimisations, since they abstract out aspects that in our particular case may be concrete.

For example, in the case of Marlowe's static analysis, we initially implemented a *symbolic* associative map primitive with the only limitation that it needed a concrete bound in the number of elements. This is straightforward to realise using the symbolic implementation of list and tuple, both provided by SBV. However, because we assumed keys were symbolic, looking up a single element required constraints that compared the element with every key up to the maximum capacity of the associative map.

Nevertheless, in Marlowe we know the values of all the keys that we are going to use in maps, because the contract is concrete, and only `Account`, `Choice`, and `Let` identifiers that are mentioned in the contract will ever make it into the `State`. So we do not need keys of the associative map to be symbolic, we can use a concrete associative map with symbolic values.

Reducing the search space by normalising parameters. The higher the number of degrees of freedom of the input, the larger the search space, and the higher the load we are putting on the SMT solver. But, if two or more different inputs have the same effect on the property we only need to include one of them.

For example, in the case of Marlowe's static analysis, Marlowe allows several inputs to be combined into a single transaction. This functionality is important because each transaction requires the issuer to pay fees. On the other hand, it also means that static analysis must consider many more possibilities, since the number of ways of partitioning inputs is exponential in the number of inputs.

However, we can devise a normal form for input traces, in which there is a maximum of one input per transaction. We only need to make sure that, for every trace, if it produces a warning, there exists a trace with only one input per transaction that also produces a warning. Using the automated proof assistant Isabelle [15], we have shown that, indeed, splitting transactions into transactions

[4] `https://github.com/input-output-hk/marlowe/blob/master/isabelle/CloseSafe.thy` (last visited on 04/04/2020)

with single inputs and the same slot interval as the original transactions does not modify the effect of those transactions on a contract[5].

This optimization reduces the search space, but transactions may still have either one or zero inputs, so there are still many ways of distributing the inputs in transactions. We explain how we reduced the search space further in Section 4.2.

4.2 Heavyweight modifications.

When optimising, if our solution is a local minimum, small changes to the parameters will not grant any improvement to the result. For that reason, in this section, we explore ways of optimising that may imply considerable rewriting of our properties, constraints, and static analysis implementation in general.

Unfortunately, not following the concrete implementation closely is much more error prone, since there are many more assumptions that we need to make and reason about. In Section 5, we explore ways of mitigating this issue.

Reducing the search space by using normalised execution paths relevant to the property. Instead of modelling the execution symbolically, we can focus on modelling the property. We do not even need to consider the representation of the counterexamples (we will discuss that in the next section), but only in what are the conditions for the property to be false.

For example, the main property we want to check is whether there is any possible execution that produces a failed payment. Thus, we only write constraints for executions in which this can happen instead of modelling all possibilities. The most complicated construct in terms of execution is the `When` construct, since it allows for transactions that are separate in time to have different effects depending on when they are issued, all other constructs will get resolved atomically in one way or another. Without loss of generality, we can structure possible executions as shown in Figure 1: we can conceptually break the contract tree into subtrees, where each subtree has a `When` construct as its root, with the exception of the subtree at the root of the original tree.

Each level of subtrees corresponds to a potential transaction, i.e: the root subtree will correspond to the first transaction, the set of subtrees that are children of the first subtree (in the original tree) will correspond to the second transaction, and so on. There may be paths which require fewer transactions/subtrees because they traverse fewer `When` constructs.

We split transactions like this because if the maximum slot number of a transaction is lower than the slot number in the timeout of a `When` it will stop before executing that `When`. Since the `When` and its continuation may be executed by a different transaction, the slot numbers for that segment of execution may be different, which means the values of `Value`s may be different, which means the amounts in payments and deposits may be different, and thus the warnings issued may be different as if both segments were executed by the same transaction.

[5] `https://github.com/input-output-hk/marlowe/blob/master/isabelle/SingleInputTransactions.thy` (last visited on 29/04/2020)

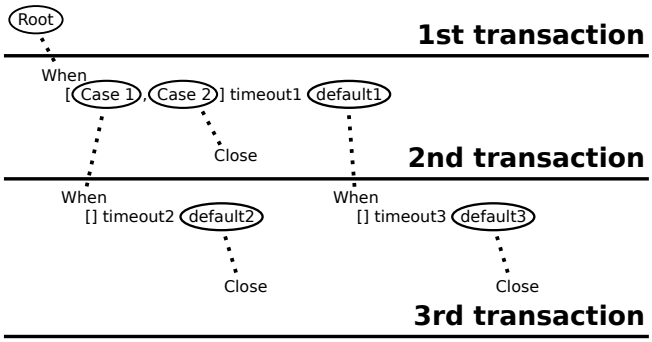

Fig. 1. Distribution of transactions with respect to a contract

However, there is an edge case: a transaction may execute past a `When` if the minimum slot number in the transaction is greater or equal than the timeout of the `When`. If the transaction expires the `When` we can no longer separate the execution before the `When` and after the `When` into two different transactions, because we know that the timeout branch of the `When` will be executed as part of the first transaction, and thus it will use the slot interval of the first transaction (same used for the first segment).

This is not only true for the timeout branch, if we include an input that is allowed by a `When`, then the first transaction will continue into the corresponding `When` branch, and we will not use a second transaction.

We cannot just constrain the maximum slot number of the first transaction to be less than the timeout in the `When` because, if we do that, then we will miss executions where one transaction expires several `When` in a row, or where it expires one `When` and provides the right input for a later `When`. And we cannot just not constrain the maximum slot number of the first transaction because then we will be considering impossible executions.

We get around this problem by allowing the slot numbers of the first and second transactions to be equal. If this happens, we will find out that one of the transactions will be marked as `UselessTransaction` by the semantics when we look at the counterexample. So we just need to filter out all transactions that produce `UselessTransaction` warnings in the final result.

To sum up, in Section 4.1, we already limited the number of inputs per transaction to a maximum of one. But now we have also assigned each of the transactions to a part of the contract, so we no longer need a symbolic list of transactions, we can use a finite concrete list of symbolic transactions.

One detail we found out during testing is that, even though a `When` belongs to the beginning of a transaction, the environment used by the `Observation`s in the `Notify` cases of the `When` correspond to the `State` before the `When` is executed (except for the slot interval). The same is true for the `Value` in the `Deposit` cases, since the amount to deposit must be calculated without considering the effects of the deposit itself.

Minimizing the representation of inputs and outputs. When we initially implemented the efficient version of static analysis for Marlowe, we did not pay any attention to the inputs and outputs. The first version would simply take a concrete `Contract` as input, and it would return a symbolic Boolean that determined whether the `Contract` was valid or not. However, if a `Contract` turns out to be invalid, we will also want to know why, so we later modified the property to give a counterexample that illustrated what went wrong. The original implementation still used some intermediate symbolic variables, but they were anonymous, and they were created during the exploration of the contract.

A simple way of obtaining a counterexample is to modify the output of the function to return the offending trace using the symbolic `Maybe` type. Surprisingly, this change increases the time required by the symbolic analysis severalfold.

An efficient way of extracting counterexample information was to pass a symbolic fixed-size list, as input to the static analysis, where each element corresponds to a transaction and consists of a tuple with four symbolic integers:

1. An integer representing the minimum slot
2. An integer representing the maximum slot
3. An integer representing the `When` case whose input is being included in the transaction, where zero represents the timeout branch (and no input).
4. An integer representing the amount of money or tokens (if the input is a `Deposit`), or the number chosen (if the input is a `Choice`)

For short branches (that require fewer transactions) we pad the end of the list with dummy transactions with all four numbers set to -1.

In order to translate this sequence of numbers into a proper list of transactions that is human readable and we can use to report the counter example, we need to use the concrete semantics together with the information obtained from the static analysis to *fill the gaps*, by iterating through the list and looking at the evolution of the contract with each transaction. This provides us with the rest of necessary information, such as the type of a transaction input (e.g: whether it is a `Deposit` or a `Choice`). We also use this process to filter transactions with no effect, i.e: they produce `UselessTransaction` as we mentioned in Section 4.2.

We also use this separation of concerns between static analysis and concrete semantics as an opportunity for applying property based testing.

5 Testing for consistency and equivalence

The more different the static analysis implementation and the concrete implementation are, the harder it is to ensure they are consistent with each other. To ensure that heavyweight optimisations remain consistent, we combine the use of automated proof assistants and the use of property based testing.

5.1 Testing for consistency

If our static analysis does not replicate all the functionality of the semantics, we can use potential discrepancies as an opportunity for testing, as shown in

Figure 2. We generate random contracts and we apply the static analysis to them in order to try to find a counterexample that produces warnings. If we cannot find any counterexamples then the test passes, but if we find one, we can test it on the semantics and see whether the counterexample indeed produces warnings in the semantics too, if it does not we have found a problem in either the static analysis or the semantics.

A limitation of this approach is that it only tests for false positives; false negatives can be detected by testing for equivalence (see Section 5.2).

In addition, we can add assertions to the process. In the case of Marlowe, if the counterexample causes errors during the execution or is formed incorrectly, it would also mean that there is a problem with the static analysis. For example, it may be that the counterexample refers to a `Case` of a `When` that does not exist, or that it has invalid or ambiguous intervals. If it has `UselessTransactions` that is ok, because we are doing that on purpose, as we mentioned in Section 4.2.

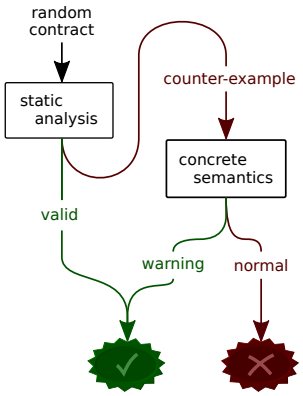

Fig. 2. Testing for consistency

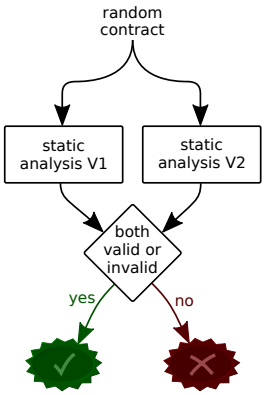

Fig. 3. Testing for equivalence

5.2 Testing for equivalence

Given two implementations of the static analysis, we have another opportunity for testing. We have one efficient implementation that is very different from the semantics and one inefficient implementation that is much closer to the semantics, and we can compare the results of the two, as shown in Figure 3.

We generate random contracts using a custom QuickCheck [4] generator, feed them to both implementations, and compare the results. If the results are the same then test passes, if they are different then one of the implementations is wrong. This approach covers both types of errors, i.e: false positives and false negatives (for both of the implementations), but the execution time of the tests is bounded from below by the slower of the two implementations. The consistency approach is thus more efficient in finding false positives.

6 Measurements

In our experiments, the heavyweight optimisations considerably reduced the requirements of both processing time and memory for Marlowe's static analysis. We present below the results of measuring the performance of static analysis on four example contracts, before[6] and after[7] the heavyweight optimisations.

Unfortunately, at the time of writing, we do not have a completely unoptimised version of the static analysis that we can use to compare, because the semantics of Marlowe have changed since we implemented that version. However, our impression from manual testing is that the impact of lightweight optimisations was much more modest than the impact of heavyweight optimisations.

Using the `perf` tool [1], we measured the execution time[8] and the overhead of the generation of the constraints and their solution by Z3 [6]. We also measured the peak RAM usage of the whole process using GNU's `time` tool [10].

In all cases the implementation with the heavyweight optimisations performs much better and scales further. In the case of the *auction* and *crowdfunding* contracts, whose size grows exponentially, both approaches quickly overwhelm the resources available in the execution environment. In the case of *rent* and *coupon bond* contracts, which grow linearly, when using the lightweight version the problem becomes intractable much faster than with the heavyweight one.

Another conclusion that can be derived from the experiments is that, in the version with lightweight optimisations, most of the processing time seems to be spent generating of the constraints, and solving is done relatively quickly by Z3; while the opposite happens for the version with heavyweight optimisations.

Auction contract – `Auction.hs`

Num. participants	1	2	3	4	5
Contract size (chars)	275	3,399	89,335	4,747,361	413,784,559
Lightweight optimisations					
Execution time	0.2205s	4m 45.2015s	N/A	N/A	N/A
Generation overhead	76.61%	96.40%	N/A	N/A	N/A
× execution time	0.1689s	4m 34.9342s	N/A	N/A	N/A
Z3 overhead	23.39%	3.60%	N/A	N/A	N/A
× execution time	0.0516s	10.2673s	N/A	N/A	N/A
RAM usage peak	44,248KB	1,885,928KB	N/A	N/A	N/A
Heavyweight optimisations					
Execution time	0.01198s	0.0289s	1.1138s	1h 5m 45.1377s	N/A
Generation overhead	16.27%	29.64%	24.09%	80.86%	N/A
× execution time	0.001949s	0.008566s	0.268314s	53m 10.038344s	N/A
Z3 overhead	83.73%	70.36%	75.91%	19.14%	N/A
× execution time	0.010031s	0.020334s	0.845486s	12m 35.099356s	N/A
RAM usage peak	17,020KB	17,740KB	49,668KB	2,364,500KB	N/A

[6] `https://github.com/input-output-hk/marlowe/blob/master/src/Language/Marlowe/Analysis/FSSemantics.hs` [last visited 19-05-2020]

[7] `https://github.com/input-output-hk/marlowe/blob/master/src/Language/Marlowe/Analysis/FSSemanticsFastVerbose.hs` [last visited 19-05-2020]

[8] The experiments were run on a laptop computer with a i9-9900K (3.6GHz) processor and two modules of 16GB of SODIMM DDR4 RAM at 2400MHz.

Crowdfunding contract – `CrowdFunding.hs`

Num. participants	1	2	3	4	5
Contract size (chars)	857	12,704	364,824	19,462,278	1,690,574,798
Lightweight optimisations					
Execution time	0.6298s	50m 53.4597s	N/A	N/A	N/A
Generation overhead	85.02%	99.82%	N/A	N/A	N/A
× execution time	0.5356s	50m 47.9635s	N/A	N/A	N/A
Z3 overhead	14.98%	0.18%	N/A	N/A	N/A
× execution time	0.0943s	5.4962s	N/A	N/A	N/A
RAM usage peak	111,980KB	5,641,056KB	N/A	N/A	N/A
Heavyweight optimisations					
Execution time	0.0125s	0.041s	1.0515s	32m 15.4478s	N/A
Generation overhead	16.68%	25.36%	37.23%	69.83%	N/A
× execution time	0.0021s	0.0104s	0.3915s	22m 31.5232s	N/A
Z3 overhead	83.32%	74.64%	62.77%	30.17%	N/A
× execution time	0.0104s	0.0306s	0.6600s	9m 43.9246s	N/A
RAM usage peak	17,016KB	18,768KB	62,108KB	3,715,020KB	N/A

Rent contract – `Rent.hs`

Num. months	1	2	3	4	5
Contract size (chars)	339	595	852	1,109	1,366
Lightweight optimisations					
Execution time	0.2850s	3.0303s	2m 53.2458s	3h 22m 13.0122s	N/A
Generation overhead	77.55%	91.25%	99.18%	99.94%	N/A
× execution time	0.2210s	2.7651s	2m 51.8252s	3h 22m 5.7324s	N/A
Z3 overhead	22.45%	8.75%	0.82%	0.06%	N/A
× execution time	0.0640s	0.2652s	1.4206s	7.2798s	N/A
RAM usage peak	42,052KB	221,960KB	1,237,092KB	9,160,616KB	N/A
Heavyweight optimisations					
Execution time	0.0114s	0.0111s	0.01132s	0.01124s	0.01255s
Generation overhead	11.55%	11.95%	13.65%	13.69%	21.00%
× execution time	0.0013s	0.0013s	0.0015s	0.0015s	0.0026s
Z3 overhead	88.45%	88.05%	86.35%	86.31%	79.00%
× execution time	0.0101s	0.0098s	0.0098s	0.0097s	0.0099s
RAM usage peak	15,536KB	15,364KB	15,400KB	15,364KB	15,540KB

Coupon bond contract - `CouponBond.hs`

Num. months	2	3	4	5	6
Contract size (chars)	479	635	791	947	1,103
Lightweight optimisations					
Execution time	0.8293s	5.8887s	1m 35.3930s	26m 36.4585s	9h 50m 22.3418s
Generation overhead	76.57%	74.91%	89.29%	72.34%	76.66%
× execution time	0.6350s	4.4112s	1m 25.1764s	19m 14.8781s	7h 32m 34.7672s
Z3 overhead	23.43%	25.09%	10.71%	27.66%	23.34%
× execution time	0.1943s	1.4775s	10.2166s	7m 21.5804s	2h 17m 47.5746s
RAM usage peak	74,180KB	209,724KB	940,924KB	3,283,384KB	13,483,908KB
Heavyweight optimisations					
Execution time	0.0092s	0.0095s	0.0097s	0.0102s	0.0105s
Generation overhead	14.51%	15.50%	15.51%	19.71%	19.69%
× execution time	0.0013s	0.0015s	0.0015s	0.0020s	0.0021s
Z3 overhead	85.49%	84.50%	84.49%	80.29%	80.31%
× execution time	0.0079s	0.0080s	0.0082s	0.0082s	0.0085s
RAM usage peak	15,636KB	15,816KB	15,788KB	15,780KB	15,760KB

These results suggest that the SBV library is able to generate constraints in a way that they are handled efficiently by Z3, but the process itself can be costly. However, the execution time required by Z3 is also lower in the case of the heavyweight optimisations as well as growing more slowly, which suggests that the optimisations described here affect both parts of the process.

7 Related work

Work in [14] documents a similar effort to ensure correctness of control software in Haskell using the SBV library; the authors also discuss performance of the analysis and apply this approach to non-functional requirements.

The idea of using constraint solvers for finding bugs is not new, and there have been a number of initiatives that have explored its application to the verification of assertions in programs written using general purpose programming languages [8,9]; as well as for the compliance with protocols [2,17].

[11] also applies constraint solvers for detecting problems in the usage of DSLs. The authors observe that SMT solvers have limited support for non-linear constraints such as exponentiation. This problem does not affect the current design of Marlowe because it does not support multiplication by arbitrary variables, and because all inputs are integer and bounded finitely.

8 Conclusion

In this paper we have summarized our work on optimising the static analysis for Marlowe contracts. We have seen that there are two distinct approaches to static analysis using SMT, both have advantages and disadvantages. One is less error prone and straightforward, but inefficient and hard to test; the other is much more efficient, versatile, and testable, but more error prone. We have also seen that many specific properties and restrictions characteristic of the target DSL can be utilized both as optimisation opportunities and, in our case, for completeness of the analysis. Symbolic execution of a Turing-complete language, would be intractable, and would require us to manually set a bound; but this is not in the case for Marlowe.

In the end, we have illustrated how to counteract the main disadvantage of the optimised approach – its propensity to errors – by using property based testing. This way we have obtained a static analysis implementation that is efficient, versatile, testable, and reliable. On the other hand, for the static analysis of Marlowe contracts, we found out that when running statistics on the equivalence testing property, most of the bugs were false negatives in the straightforward implementation, and the optimised implementation seems to be more reliable thanks to the consistency tests that we run beforehand.

Another advantage of the optimised implementation is that, because it relies on fewer and simpler features, it is compatible with more SMT solvers which, in turn, means that it is less reliant on the correctness or efficiency of a single

solver. If one solver fails to give an answer, we can try another; if we want further evidence that a contract is valid, we can test it with several solvers.

In the future, we would like to extend static analysis to cover other potential problems in Marlowe contracts and to aid their development. We plan to use static analysis to locate unreachable subcontracts, to allow developers to provide custom assertions and check their satisfiability, and to allow users to inspect the possible maximum and minimum values that particular expressions can reach.

References

1. Performance analysis tools for Linux. `https://github.com/torvalds/linux/tree/master/tools/perf` [last visited 20-05-2020]
2. Ball, T., Rajamani, S.K.: Automatically validating temporal safety properties of interfaces. In: SPIN Workshop on Model Checking of Software. Springer (2001)
3. Barrett, C., Stump, A., Tinelli, C., et al.: The SMT-LIB standard: Version 2.0. In: Proc. 8th International Workshop on Satisfiability Modulo Theories (2010)
4. Claessen, K., Hughes, J.: QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP **46** (2000)
5. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer (2004)
6. De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Internat. Conf. on Tools and Algorithms for the Construction and Analysis of Systems. Springer (2008)
7. Erkök, L.: SBV: SMT based verification in Haskell. Software library (2019)
8. Gulwani, S., Srivastava, S., Venkatesan, R.: Program analysis as constraint solving. In: Proceedings of the 29th ACM SIGPLAN Conference PLDI (2008)
9. Jackson, D., Vaziri, M.: Finding bugs with a constraint solver. ACM SIGSOFT Software Engineering Notes **25**(5) (2000)
10. Keppel, D., MacKenzie, D., Juul, A.H., Pinard, F.: GNU time tool. `https://www.gnu.org/software/time/` [last visited 20-05-2020] (1998)
11. Keshishzadeh, S., Mooij, A.J., Mousavi, M.R.: Early fault detection in DSLs using SMT solving and automated debugging. In: International Conference on Software Engineering and Formal Methods. Springer (2013)
12. Lamela Seijas, P., Nemish, A., Smith, D., Thompson, S.: Marlowe: implementing and analysing financial contracts on blockchain. Workshop on Trusted Smart Contracts, to appear (2020)
13. Lamela Seijas, P., Thompson, S.: Marlowe: Financial contracts on blockchain. In: Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice. ISoLA 2018. Springer (2018)
14. Mokhov, A., Lukyanov, G., Lechner, J.: Formal verification of spacecraft control programs (experience report). In: Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell (2019)
15. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: a proof assistant for higher-order logic, vol. 2283. Springer Science & Business Media (2002)
16. Sheard, T., Jones, S.P.: Template meta-programming for Haskell. In: Proceedings of the 2002 ACM SIGPLAN workshop on Haskell (2002)
17. Xie, Y., Aiken, A.: Saturn: A SAT-based tool for bug detection. In: International Conference on Computer Aided Verification. Springer (2005)