Verifying Smart Contracts with the Move Prover

David Dill[1], Wolfgang Grieskamp[1],
Junkil Park[1], Shaz Qadeer[1], Meng Xu[1], and Emma Zhong[1]

Facebook

Abstract. TBD.

Keywords: First keyword · Second keyword · Another keyword.

1 Introduction

Smart contracts have been for longer considered a promising domain to employ formal verification [34]. Smart contracts deal with computations which require highest correctness assurance, as they manipulate assets of large value in an irrevocable way. Those computations need to follow a growing set of rules, reflecting a large diversity of regulations in place, and therefore are vulnerable to bugs.

On the other hand, from the viewpoint of verification techniques, smart contracts are relatively *easier* to verify than general programs, because they exist in a well-defined, isolated execution environment. Computations are typically sequential, not i/o-bound, and deterministic. These basic advantages of smart contracts can be further augmented by choice of a verification friendly language and virtual machine design.

In this paper we report about the *Move prover* (abbreviated MVP), a tool for the *Move* smart contract language [6]. Move has been developed together with the Diem blockchain [30], and has been designed from the beginning with verification in mind. The language supports specifying pre, post, and aborts conditions of functions, as well as invariants over data structures and over the content of the global persistent memory (which represents programmable "storage" in smart contracts). Specification constructs include universal and existential quantification over arbitrary data types and are therefore generally not decidable.

Despite this specification richness, MVP is capable of verifying the full Diem framework [31], the Move implementation of the Diem blockchain [30]. The framework provides functionality for managing accounts and their interaction, including multiple currencies, account roles, and rules for transactions. It consists of approximately 12,000 lines of Move program code and specifications. The framework is exhaustively specified, and *verification runs fully automated alongside with unit and integration tests*, which we consider a significant practical result for formal verification adoption.

From the point of the first Move Prover publication in [39], many improvements have been made to make such usage possible. Those are along speed, predictability,

error reporting, and absence of false positives and timeouts. We developed a number of novel translation techniques which optimized SMT performance and, more importantly, resulted in generally more predictable behavior. While in previous versions of the tool, we saw timeouts frequently, plagued by the Butterfly effect [17], the current version only rarely runs into those problems.

This paper is organized as follows. We first give an introduction into the Move language and how MVP is used with it. We then discuss in more detail the design of MVP, and the most important translation techniques it uses, including elimination of references from Move programs, evaluating global memory invariants by injection them at the right places into the code, monomorphization of generic programs, and modular verification. For furthergoing study, the appendix discusses injection of function specifications, and the mapping to the Boogie intermediate verification language [3].

2 Move and the Prover

Move was developed for the Diem blockchain [30], but its design is not specific to blockchains. A Move execution consists of a sequence of updates evolving a *global persistent memory state*, which we just call the *(global) memory*. Updates are executed in a *transactional* style.

The global memory is organized as a collection of resources, described by Move structures (data types). A resource in memory is indexed by a pair of a type (possibly instantiated) and an address (for example the address of a user account). For instance, the expression `exists<Coin<USD>>(addr)` will be true if there is a value of type `Coin<USD>` stored at `addr`. As seen in this example, Move uses type generics, and working with generic functions and types is rather idiomatic for Move. Notice that account addresses are not just arbitrary values but have a specific role in Move's programming methodology related to access control via the builtin type of *signers*, as will be discussed later.

A Move application consists of a set of *transaction scripts*. Each of those script defines a Move function with input parameters but no output parameters. This function updates the global memory, including emitting events. The execution of this function can fail via a well-defined abort mechanism, in which case the memory stays unmodified. An environment emits a sequence of calls to such scripts, thereby evolving the memory.

Programming in Move In Move, one defines transactions via so-called *script functions* which take a set of parameters. Those functions can call other functions. Script and regular functions are encapsulated in *modules*. Move modules are also the place where structs are defined. An illustration of a Move contract is given in Fig. 1 (for a more complete description see the Move Book [32]). The example is a simple account which holds a balance, defined in the script function `transfer`. Scripts get passed in so called *signers* which are tokens which represent an authorized account address. The caller of the script – an external program – has ensured that the owner of the signer account address has agreed to execute this script. Notice that in the

Fig. 1: Account Example Program

```
module Account {
  struct Account has key {
    balance: u64,
  }

  fun withdraw(account: address, amount: u64) acquires Account {
    let balance = &mut borrow_global_mut<Account>(account).balance;
    assert(*balance >= amount, Errors::limit_exceeded());
    *balance = *balance - amount;
  }

  fun deposit(account: address, amount: u64) acquires Account {
    let balance = &mut borrow_global_mut<Account>(account).balance;
    assert(*balance <= Limits::max_u64() - amount, Errors::limit_exceeded());
    *balance = *balance + amount;
  }

  public(script) fun transfer(from: &signer, to: address, amount: u64)
  acquires Account {
    assert(Signer::address_of(from) != to, Errors::invalid_argument());
    withdraw(Signer::address_of(from), amount);
    deposit(to, amount);
  }
}
```

code, the `assert` statement causes a Move transaction to abort execution if the condition is not met. Abortion can also happen implicitly; for example, the expression `borrow_global_mut<T>(addr)` will abort if no resource `T` exists at `addr`.

Specifying in Move The specification language supports *Design By Contract* [23]. Developers can provide pre and post conditions for functions, which include conditions over (mutable) parameters and global memory. Developers can also provide invariants over data structures, as well as the (state-dependent) content of the global memory. Universal and existential quantification both over bounded domains (like the indices of a vector) as well of unbounded domains (like all memory addresses, all integers, etc.) are supported. The latter makes the specification language very expressive, but also renders the verification problem in theory undecidable (and in practice dependent on heuristic decision procedures).

Fig. 2 illustrates the specification language by extending the account example in Fig. 1 (for the definition of the specification language see [33]). This adds the specification of the `transfer` function, a helper function `bal` for use in specs, and two global memory invariants. The first invariant states that a balance can never drop underneath a certain minimum. The second invariant refers to an update of global memory with pre and post state: the balance on an account can never decrease in one step more than a certain amount. Note that while the Move programming language has only unsigned integers, the specification language uses arbitrary precision signed integers, making it convenient to specify something like `x + y <= limit`, without need to worry about overflow.

Fig. 2: Account Example Specification

```
module Account {
  spec transfer {
    let from_addr = Signer::address_of(from);
    aborts_if from_addr == to;
    aborts_if bal(from_addr) < amount;
    aborts_if bal(to) + amount > Limits::max_u64();
    ensures bal(from_addr) == old(bal(from_addr)) - amount;
    ensures bal(to) == old(bal(to)) + amount;
  }

  spec fun bal(acc: address): u64 {
    global<Account>(acc).balance
  }

  invariant forall acc: address where exists<Account>(acc):
    bal(acc) >= AccountLimits::min_balance();

  invariant update forall acc: address where exists<Account>(acc):
    old(bal(acc)) - bal(acc) <= AccountLimits::max_decrease();
}
```

Specifications for the `withdraw` and `deposit` functions have been omitted in this examples. MVP supports omitting specs for non-recursive functions, in which case they are treated as being inlined at caller site.

A discerning reader may have noted that the program in Fig. 1 does not actually satisfy the specification in Fig. 2. This will be discussed in the next section.

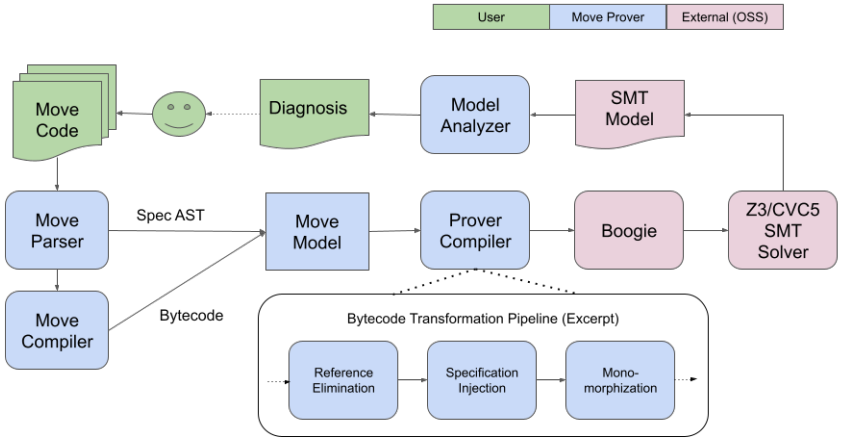
Running the Prover MVP operates fully automated, quite similar to a type checker or linter, and is expected to conclude in reasonable execution time, so it can be integrated in the regular development workflow. Running MVP on the `module Account` produces multiple errors. The first is this one:

```
error: abort not covered by any of the ‘aborts_if‘ clauses
   -- account.move:24:3
   |
13 |       let balance = &mut borrow_global_mut<Account>(account).balance;
   |                          ----------------- abort happened here
   |
   =     at account.move:18: transfer
   =         from = signer{0x18be}
   =         to = 0x18bf
   =         amount = 147u8
   =     at ...
```

MVP detected that an implicit aborts condition is missing in the specification of the `withdraw` function. It prints the context of the error, as well as an *execution trace* which lead to the error. Values of variable assignments from the counterexample found by the SMT solver are printed together with the execution trace. Logically, the counter example presents an instance of assignments to variables such that program and specification disagree. In general, MVP attempts to produce diagnostics readable for Move developers without the need of understanding any internals of the prover.

The next errors produced are about the memory invariants in Fig. 2. Both of them do not hold; we show only the first failure:

Fig. 3: Move Prover Architecture




```
error: global memory invariant does not hold
   -- account.move:39:3
   |
39 |     invariant forall acc: address where exists<Account>(acc):
40 |       bal(acc) >= AccountLimits::min_balance();
   |
   =     at account.move:6: withdraw
   =         account = 0x0
   =         amount = 11u8
   =     at account.move:7: withdraw
   =         balance = &15u8
   =     at ...
```

This happens because in the program in Fig. 1, we did not made any attempt to respect the limits in `min_balance()` and `max_decrease()`. We leave it open here (but covered it in App. B) how to fix this problem, which requires to add some more `assert` statements to the code and abort if the limits are not met.

In reality, the programs and specifications MVP deals with are significant larger than in this example. For example, the conditions under which a transaction in the Diem framework can abort typically has dozens of individual predicates, stemming from other functions called by this transaction. Moreover, there are hundreds of memory invariants specified, encoding access control and other requirements for the Diem blockchain.

3 Move Prover Design

The architecture of MVP is illustrated in Fig. 3. Move code (containing specifications) is given as input to the Move tool chain, which produces two artifacts: the abstract syntax tree (AST) of the specifications, and the generated bytecode. The *Move Model* merges both bytecode and specifications, as well as other metadata from the original code, into a unique object model which is input to the remaining tool chain.

The next phase is the actual *Prover Compiler*, which is implemented as a pipeline of bytecode transformations. Only an excerpt of the most important transformations is shown (Reference Elimination, Specification Injection, and Monomorphization). These transformations will be conceptually described in more detail in subsequent sections. While they happen in reality on an extended version of the bytecode, we will illustrate them on a higher level of abstraction, as Move source level transformations.

The transformed bytecode is next compiled into the Boogie intermediate verification language [3]. Boogie supports an imperative programming model which is well suited for the encoding of the transformed Move code. Boogie in turn can translate to multiple SMT solver backends, namely Z3 [25] and CVC5 [29]; the default choice for the Move prover is currently Z3.

Subsequently, we will focus on the major bytecode transformations.

3.1 Reference Elimination

The Move language supports references to data stored in global memory and on the stack. Those references can point to interior parts of the data. The reference system is based on *borrow semantics* [8] as it is also found in the Rust programming language. One can create (immutable) references `&x` and mutable references `&mut x`, and derive new references by field selection (`&mut x.f` and `&x.f`). The borrow semantics of Move provides the following guarantees (ensured by the borrow checker [7]):

- For any given location in global memory or on the stack, there can be either exactly one mutable reference, or n immutable references. Hereby, it does not matter to what interior part of the data is referred to.
- Dangling references to locations on the stack cannot exist; that is, the lifetime of references to data on the stack is restricted to the lifetime of the stack location.

These properties enable us to *effectively eliminate* references from the Move program, reducing the verification complexity significantly, as we do not need to reason about sharing. It comes as no surprise that the same discipline of borrowing which makes Move (and Rust) programs safer by design also makes verification simpler.

Immutable References Since during the existence of an immutable reference no mutation on the referenced data can occur, we can simply replace references by the referred value.

An example of the applied transformation is shown below. We remove the reference type constructor and all reference-taking operations from the code:

```
fun select_f(s: &S): &T { &s.f } ⇝ fun select_f(s: S): T { s.f }
```

Notice that at Move execution time, immutable references serve performance and ownership objectives (avoid copies); however, the symbolic reasoning engines we use have a different representation of values, in which structure sharing is common and copying is cheap.

Mutable References Each mutation of a location `l` starts with an initial borrow for the whole data stored in this location (in Move, `borrow_global_mut<T>(addr)` for global memory, and `&mut x` for a local on the stack). Let's call the reference resulting from such a borrow `r`. As long as this reference is alive, Move code can either update its value (`*r = v`), or derive a sub-reference (`r' = &mut r.f`). The mutation ends when `r` (and the derived `r'`) go out of scope. Because of the guarantees of the borrow semantics, during the mutation of the data in `l` no other reference can exist into data in `l`.

The fact that `&mut` has exclusive access to the whole value in a location allows to reduce mutable references to a *read-update-write* cycle. One can create a copy of the data in `l` and single-thread it to a sequence of mutation steps which are represented as purely functional data updates. Once the last reference for the data in `l` goes out of scope, the updated value is written back to `l`. This effectively turns an imperative program with references into an imperative program which only has state updates on global memory or variables on the stack, a class of programs which is known to have a significantly simpler semantics. We illustrate the basics of this approach by an example:

```
fun increment(x: &mut u64) { *x = *x + 1 }
fun increment_field(s: &mut S) { increment(&mut s.f) }
fun caller(): S { let s = S{f:0}; update(&mut s); s }
⇝
fun increment(x: u64): u64 { x + 1 }
fun increment_field(s: S): S { s[f = increment(s.f)] }
fun caller(): S { let s = S{f:0}; s = update(s); s }
```

Dynamic Mutable References While the setup in above example covers a majority of the use cases in every day Move code, there are more complex ones to consider, namely that the value of a reference depends on runtime decisions. Consider the following Move code:

```
let r = if (p) &mut s1 else &mut s2;
increment_field(r);
```

Additional information in the logical encoding is required to deal with such cases. At the execution point a reference goes out of scope, we need to know from which location it was derived, so we can write back the updated value correctly. Fig. 4 illustrates the approach for doing this. A new Move prover internal type `Mut<T>` is introduced which carries the location from which `T` was derived together with the value. It supports the following operations:

- `Mvp::mklocal(value, LOCAL_ID)` creates a new mutation value for a local with the given local id. Local ids are transformation generated constants kept opaque here.
- Similarly, `Mvp::mkglobal(value, TYPE_ID, addr)` creates a new mutation for a global with given type and address. Notice that in the current Move type system, we would not need to represent the address, since there can be only one mutable reference into the entire type (via the acquires mechanism). However, we keep this more general here, as the Move type system might change.

Fig. 4: Elimination of Mutable References

```
fun increment(x: &mut u64) { *x = *x + 1 }
fun increment_field(s: &mut S) {
  let r = if (s.f > 0) &mut s.f else &mut s.g;
  increment(r)
}
fun caller(p: bool): (S, S) {
  let s1 = S{f:0, g:0}; let s2 = S{f:1, g:1};
  let r = if (p) &mut s1 else &mut s2;
  increment_field(r);
  (s1, s2)
}
⇝
fun increment(x: Mut<u64>): Mut<u64> { Mvp::set(x, Mvp::get(x) + 1) }
fun increment_field(s: Mut<S>): Mut<S> {
  let r = if (s.f > 0) Mvp::field(s.f, S_F) else Mvp::field(s.g, S_G);
  r = increment(r);
  if (Mvp::is_field(r, S_F))
    s = Mvp::set(s, Mvp::get(s)[f = Mvp::get(r)]);
  if (Mvp::is_field(r, S_G))
    s = Mvp::set(s, Mvp::get(s)[g = Mvp::get(r)]);
  s
}
fun caller(p: bool): S {
  let s1 = S{f:0, g:0}; let s2 = S{f:1, g:1};
  let r = if (p) Mvp::mklocal(s1, CALLER_s1)
          else Mvp::mklocal(s2, CALLER_s2);
  r = increment_field(r);
  if (Mvp::is_local(r, CALLER_s1))
    s1 = Mvp::get(r);
  if (Mvp::is_local(r, CALLER_s2))
    s2 = Mvp::get(r);
  (s1, s2)
}
```

- With `r' = Mvp::field(r, FIELD_ID)` a mutation value for a sub-reference is created for the identified field.
- The value of a mutation is replaced with `r' = Mvp::set(r, v)` and retrieved with `v = Mvp::get(r)`.
- With the predicate `Mvp::is_local(r, LOCAL_ID)` one can test whether `r` was derived from the given local, and with `Mvp::is_global(r, TYPE_ID, addr)` whether it was derived from the specified global. `Mvp::is_field(r, FIELD_ID)` tests whether it is derived from the given field.

The Move Prover has a partial implementation of the illustrated transformation. The completeness of this implementation has not yet been formally investigated, but we believe that it covers all of Move, with the language's simplification that we do not need to distinguish addresses in global memory locations.[1] (See discussion of `Mvp::mkglobal` above.) The transformation also relies on that in Move there are no recursive data types, so field selection paths are statically known. While those things can be potentially generalized, we have not yet investigated this direction.

[1] **TODO(wrwg)**: Need to investigate loops!

The implementation constructs a *borrow graph* from the program via a data flow analysis. The borrow graph tracks both when references are released as well as how they relate to each other: e.g. `r' = &mut r.f` creates a edge from `r` to `r'` labeled with `f`, and `r' = &mut r.g` creates another also starting from `r`. For the matter of this problem, a reference is not released until a direct or indirect borrow on it goes out of scope; notice that its lifetimes in terms of borrowing is larger than the scope of its usage. The borrow analysis is *inter-procedural* requiring computed summaries for the borrow graph of called functions.

The resulting borrow graph is then used to guide the transformation, inserting the operations of the `Mut<T>` type as illustrated in Fig 4. Specifically, when the borrow on a reference ends, the associated mutation value must be written back to its parent mutation or the original location (e.g. line 29 in Fig. 4). The presence of multiple possible origins leads to case distinctions via `Mvp::is_X` predicates; however, these cases are rare in actual Move programs.

3.2 Global Invariant Injection

The goal of global invariant injection is to ensure that invariants over global memory are enforced at any time but with the minimal number of verification steps. A naive implementation which is semantically correct could evaluate all invariants whenever the memory state changes, but this is not practical. To understand how we optimize the implementation, the concept of global invariants need to be more precisely defined.

Invariant Types and Proof Methodology *Inductive* invariants are properties declared in Move modules that must (by default) hold for the global memory at all times. Those invariants typically quantify over addresses. See Fig. 2 for examples. Based on Move's borrow semantics, inductive invariants don't need to hold while memory is mutated; this is reflected by the reference elimination described in Sec. 3.1, and based on that the state of mutation is not visible to anybody until it is written back to global memory.

Update invariants are properties which relate two states. Typically they are enforced after an update of global memory, and are able to use the `old` function to access the state before the update.

Verification of both kinds of invariants can be *suspended*. That means, instead of being verified at the time a memory update happens, they are verified at the call site of the function which updates memory. This is a feature necessary to describe invariants over dependent parts of the global memory which cannot be atomically updated. Functions with external callers (public or script functions) cannot suspend invariant verification, ensuring that integrity of verification is preserved.

Inductive invariants are proven by induction over the evolution of the global memory. The base case is that the invariant must hold in the empty state that precedes the genesis transaction. For the induction step, we can assume that the invariant holds at each verified function entry point for which it is not suspended, and now must prove that it holds after program points which are either direct updates of global memory, or calls to functions which suspend invariants.

Fig. 5: Basic Global Invariant Injection

```
fun f(a: address) {
  let r = borrow_global_mut<S>(a);
  r.value = r.value + 1
}
invariant [I1] forall a: address: global<S>(a).value > 0;
invariant [I2] update forall a: address:
    global<S>(a).value > old(global<S>(a).value);
⇝
fun f(a: address) {
  spec assume I1;
  Mvp::snapshot_state(I2_BEFORE);
  r = <increment mutation>;
  spec assert I1;
  spec assert I2[old = I2_BEFORE];
}
```

For update invariants, no soundness issue exist, as they just relate two memories. The pre-state is some memory captured before an update happens, and the post state the current state. When an update invariant is suspended, it should be designed to be transitive, such that its evaluation is not dependent from the time of pre-state capture (e.g. `invariant update [suspendable] version()>= old(version())` satisfies this property).

Modular Verification We wish to support open systems to which untrusted modules can be added to without invalidating invariants that have already been proven. For each invariant, there is a defined subset of Move modules (called a *cluster*). If the invariant is proven for the modules in the cluster, it is guaranteed to hold in all other modules – even those that were not yet defined when the invariant was proven. The cluster must contain every function that can invalidate the invariant, and, in case of invariant suspension, all callers of such a function. Importantly, functions outside the cluster can never invalidate an invariant, so those functions trivially preserve the invariant, so it is only necessary to verify functions defined in the cluster.

MVP verifies a given set of modules at a time (typically one). The modules being verified are called the *target modules*, and the global invariants to be verified are called *target invariants*, which are all invariants defined in the target modules. The cluster is then the smallest set as specified above such that all target modules are contained.

Basic Translation We first look at injection of global invariants in the absence of memory and functions with type parameters. Fig. 5 contains an example for the supported invariant types and their injection into code. The first invariant, `I1`, is an inductive invariant. It is assumed on function entry, and asserted after the state update. The second, `I2`, is an update invariant, which relates pre and post states. For this a state snapshot is stored under some label `I2_BEFORE`, which is then used in an assertion.

Global invariant injection is optimized by knowledge of the prover, obtained by static analysis, about accessed and modified memory. Let `accessed(f)` be the

Fig. 6: Global Invariant Injection and Genericity

```
invariant [I1] global<S<u64>>(0).value > 1;
invariant<T> [I2] global<S<T>>(0).value > 0;
fun f(a: address) { borrow_global_mut<S<u8>>(0).value = 2 }
fun g<R>(a: address) { borrow_global_mut<S<R>>(0).value = 3 }
⇝
fun f(a: address) {
  spec assume I2[T = u8];
  <<mutate>>
  spec assert I2[T = u8];
}
fun g<R>(a: address) {
  spec assume I1;
  spec assume I2[T = R];
  <<mutate>>
  spec assert I1;
  spec assert I2[T = R];
}
```

memory accessed by a function, and `modified(f)` the memory be modified. Let `accessed(I)` by an invariant (including transitively by all functions it calls).

- Inject `assume I` at entry to `f` *if* `accessed(f)` has overlap with `accessed(I)`.
- At every point in `f` where a memory location `M` is updated, or where a return from a function `f'` happens s.t. `M in modifies(f')` and for which `I` is suspended, inject `assert I` after the point *if* `M in accessed(I)`. Also, if `I` is an update invariant, before the update or call inject a memory snapshot save.

Genericity Generic type parameters make the problem of determining whether a function can modify an invariant more difficult. Consider the example in Fig. 6. Invariant `I1` holds for a specific type instantiation `S<u64>`, whereas `I2` is generic over all type instantiations for `S<T>`.

The non-generic function `f` which works on the instantiation `S<u8>` will have to inject the *specialized* instance `I2[T = u8]`. The invariant `I1`, however, does not apply for this function, because there is no overlap with `S<u64>`. In contrast, in the generic function `g` we have to inject both invariants. Because this function works on arbitrary instances, it is also relevant for the specific case of `S<u64>`.

In the general case, we are looking at a unification problem of the following kind. Given the accessed memory of a function `f<R>` and an invariant `I<T>`, we compute the pairwise unification of memory types. Those types are parameterized over `R` resp. `T`, and successful unification will result in a substitution for both. On successful unification, we include the invariant with `T` specialized according to the substitution.

Notice that there are implications related to monomorphization coming from the injection of global invariants; those are discussed in Sec. 3.3.

3.3 Monomorphization

Monomorphization is the process of removing all generic types from a Move program by *specializing the program for all relevant type instantiations*.

Fig. 7: Basic Monomorphization

```
struct S<T> { .. }
fun f<T>(x: T) { g<S<T>>(S(x)) }
fun g<S:key>(s: S) { move_to<S>(.., s) }
⇝
struct T{}
struct S_T{ .. }
fun f_T(x: T) { g_S_T(S_T(x)) }
fun g_S_T(s: S_T) { move_to<S_T>(.., s) }
```

Basic Monomorphization To verify a generic function, monomorphization skolemizes the type parameter into a given type. It then, for all functions which are called, inserts their code (or specification, depending on whether the called function is inlined or not) specializing it for the given type instantiation, including specialization of all used types. Fig. 7 sketches this approach.

The underlying conjecture is that if we verify `f_T`, we have also verified it for all possible instantiations. However, this statement is only correct for code which does not depend on runtime type information.

Type Dependent Code The type of genericity Move provides does not allow for full type erasure as often found in programming languages. That is because types are used to *index* global memory (e.g. `global<S<T>>(addr)` where `T` is a generic type). Consider the following Move function:

```
fun f<T>(..) { move_to<S<T>>(s, ..); move_to<S<u64>>(s, ..) }
```

Depending on how `T` is instantiated, this function behaves differently. Specifically, if `T` is instantiated with `u64` the function will always abort at the second `move_to`, since the target location is already occupied.

The important property enabling monomorphization in the presence of type dependent code is that one can identify the situation by looking at the memory accessed by code and injected specifications. From this one can derive *additional instantiations of the function* which need to be verified. For the example above, verifying both `f_T` and an instantiation `f_u64` will cover all relevant cases of the function behavior. Notice that this treatment of type dependent code is specific to the problem of verification, and cannot directly be applied to execution.

The algorithm for computing the instances which require verification works as follows. Let `f<T1,..,Tn>` be a verified target function which has all specifications injected and inlined function calls expanded.

- For each memory `M in modifies(f)`, if there is a memory `M' in modifies(f)+ accessed(f)` such that `M` and `M'` can unify via `T1,..,Tn`, collect an instantiation of the type parameters `Ti` from the resulting substitution. This instantiation may not assign values to all type parameters, and those unassigned parameters stay as is. For instance, `f<T1, T2>` might have a partial instantiation `f<T1, u8>`.
- Once the set of all those partial instantiations is computed, it is extended by unifying the instantiations against each other. If `<t>` and `<t'>` are in the set, and they unify under the substitution `s`, then `<s(t)>` will also be part of the set. For example, consider `f<T1, T2>` which modifies `M<T1>` and `R<T2>`, as well as

accesses M<u64> and R<u8>. From this the instantiations <u64, T2> and <T1, u8> are computed, and the additional instantiation <u64, u8> will be added to the set.
- If after computing and extending instantiations any type parameters remain, they are skolemized into a given type as described in the previous section.

To understand the correctness of this procedure, consider the following arguments:

- *Direct interaction* Whenever a modified memory M<t> can influence the interpretation of M<t'>, a unifier must exist for the types t and t', and an instantiation will be verified which covers the overlap of t and t'.
- *Indirect interaction* If there is an overlap between two types which influences whether another overlap is semantically relevant, the combination of both overlaps will be verified via the extension step.

Notice that even though it is not common in regular Move code to work with both memory S<T> and, say, S<u64> in one function, there is a scenario where such code is implicitly created by injection of global invariants. Consider the example in Fig. 6. The invariant I1 which works on S<u64> is injected into the function g<R> which works on S<R>. When monomorphizing g, we need to verify an instance g_u64 in order to ensure that I1 holds.

4 Analysis

Predictability and Performance Improvements Compared to the version of the Move Prover which was released in September 2020 as part of version 1.0 of the Diem project we observe many improvements in verification speed and predictability of verification outcome, which we track back to the bytecode transformations we have described here. These results are, however, not so easy to quantify, as the Move language has evolved and sources written today are not compatible with sources this time back. Moreover, the Diem framework and its specification, which is the basis of our benchmarking, has evolved as well, adding more functionality and tighter specifications.

As one data point we use the DiemAccount module, the biggest module in the Diem framework. This module encapsulates basic functionality to create and maintain multiple types of accounts on the blockchain, as well as manage their coin balances. It was called LibraAccount in release 1.0 of MVP. The below table lists the number of lines, functions, invariants, conditions (requires, ensures, and aborts-if), as well as the verification times:

Module	Lines	Functions	Invariants	Conditions	Timing
LibraAccount	1975	72	10	113	**9.131s**
DiemAccount	2554	64	32	171	**6.290s**

Notice that DiemAccount has significantly grown in size compared to LibraAccount . Specifically, additional specifications have been added. Moreover, in the original

`LibraAccount`, some of the most complex functions had to be disabled for verification because the old version of MVP would timeout on them. In contrast, in `DiemAccount` and with the new version, all functions are verified. Nevertheless, verification time has been improved by roughly 30%, in the presence of 3x more global invariants, and .5x more function conditions.

We were able to observe similar improvements for the remaining of the 40 modules of the Diem framework. All of the roughly 1/2 dozen timeouts in verification in the framework resolved after introduction of the transformations described in this paper. Also, specifications which were introduced after the new transformations did not introduce new timeouts. A further improvement is that often, in cases where specification and program disagree, a timeout occurred which went away only after fixing the spec or the code, making debugging of such verification failures rather hard. This problem also disappeared.

Causes for the Improvements It is hard to clearly identify the reasons for improvements when dealing with heuristic systems like SMT solvers. Moreover, besides of the transformations described in this paper, many more smaller changes had been made to MVP, including micro-tuning the SMT encoding of the verification problem in Boogie. Tracking the causes of improvements incrementally was not the highest priority of our work.

Nevertheless, we believe that one of the biggest impacts, specifically regards removal of timeouts and predictability of verification, is monomorphization. The reason for this is that monomorphization allows a multi-sorted representation of values in Boogie (and eventually the SMT solver). In contrast, before monomorphization, we used a universal domain for values in order to represent values in generic functions, roughly as follows:

```
type Value = Num(int) | Address(int) | Struct(Vector<Value>) | ...
```

This creates a large overhead for the SMT solver, as we need to exhaustively inject type assumptions (e.g. that a `Value` is actually an `Address`), and pack/unpack values. Consider a quantifier like `forall a: address: P(x)` in Move. Before monomorphization, we have to represent this in Boogie as `forall a: Value: is#Address(a)=> P(v#Address(a))`. This quantifier is triggered where ever `is#Address(a)` is present, independent of the structure of `P`. Over-triggering or inadequate triggering of quantifiers is one of the suspected sources of timeouts, as also discussed in [17].

Moreover, before monomorphization, global memory was indexed in Boogie by an address and a type instantiation. That is, for `struct R<T>` we would have one Boogie array `[Type, int]Value`. With monomorphization, the type index is eliminated, as we create different memory variables for each type instantiation. Quantification over memory content works now on a one-dimensional instead of an n-dimensional Boogie array.

Discussion and Related Work Many approaches have applied to the verification of smart contracts [19,24,34]. A recent survey [34] distinguishes between *contract*

and *program* level approaches. Our approach has aspects of both: we address program level properties via pre/post conditions, and contract ("blockchain state") level properties via global invariants. In both cases, we use traditional predicate logic to write these properties, characterized as Hoare logic by the paper.

While [34] refers to at least two dozen systems for smart contract verification, to the best of our knowledge, the Move ecosystem is the first one where contract programming and specification language are fully integrated, and the language is designed from first principles influenced by verification. Methodologically, Move and the Move prover are therefore closer to systems like Dafny [16], or the older Spec# system [4], where instead of adding a specification approach posterior to an existing language, it is part from the beginning. This allows us not only to deliver a more consistent user experience, but also to make verification technically easier by curating the programming language, as reflected in Move's absence of dynamic dispatching and the notorious re-entrance problem [10], as well as the borrow semantics which enables optimizations like reference elimination (Sec. 3.1).

In contrast to the other approaches that only focus on specific vulnerability patterns [9,20,26,36], MVP offers the rich specification language based on program logic, thus allowing users to define the specifications of their contracts. As regards expressiveness of specification, to the best of our knowledge, no existing specification approach for smart contracts based on inductive Hoare logic has similar expressiveness. We support universal quantification over arbitrary memory content, a suspension mechanism of invariants to allow non-atomic construction of memory content, and generic invariants. The program verification approaches in Solidity [12,13,15] does not support quantifiers, because it interprets programming language constructs a specifications and has no dedicated specification language. While in Solidity one can simulate aspects of global invariants using modifiers by attaching pre/post conditions, this is not the same as our invariants, which are guaranteed to hold independent of whether a user may or (accidentally) may not attach a modifier. Moreover, from experimenting with a similar approach in Move, we know that adding invariants as pre/post conditions can be highly inefficient, because they need to be verified independent from whether a function actually changes state. In contrast, our approach to inject invariants optimizes when an invariant is actually verified. While the expressiveness of Move specifications and MVP comes with the price of undecidability and the dependency from heuristics in SMT solvers, MVP is capable to deal with this by its elaborated translation to SMT logic, as partially described in this paper. The result is a practical verification system that is fully integrated into the Diem blockchain production process, which (to the best of our knowledge) is novel by itself. Other related works on the smart contract verification employ the theoretical foundations including the framework [27] (e.g., [14]), [21] (e.g., [5,11]), and proof assistants such as Coq [28] (e.g., [37,38]).

Move has similar characteristics to Rust [22] such as borrow semantics. Borrow semantics is considered a good way to perform high-performance *and* safe programming in the Rust community. There also have been works on the formal verification of Rust programs [1,2,18,35]. Even though the kind of reference elimination we perform could also be done for the safe Rust language subset, to the best of our

knowledge, this has not been attempted before. The same technique could likely not only be used for verification, but also for runtime execution, potentially obtaining higher speed for smaller data structures by improving processor cache locality.

Open Problems and Future Work While currently MVP is operating to our satisfaction, there are multiple open problems as well. For one, the set of smart contracts we are verifying still have only medium complexity, and specifically loops are rare in the Diem framework (loops were not discussed in this paper for reasons of space; we do incorporate loop invariants for their verification). We anticipate the problem of timeouts to hit us again down the road as the number of applications of MVP grow, and further refinement of the translation will be needed. Furthermore, while currently there are no occurrences of false positives, we expect this problem to hit as well.

The biggest obstacle for wider adoption, however, is seen in the complexity of authoring and reading specifications, related to a phenomena we refer to as "specification boilerplate". Pre/post condition specifications are often verbose and difficult to read and write. At the time of this writing, function specifications in the Diem framework are often of similar size than function implementations. A common problem is also sharing between aspects of specifications: for example, if function `f` calls function `g` and the later function aborts, the aborts condition propagates from `g` to the caller `f`. While the Move specification language has mechanisms to express this kind of sharing by referring to the aborts condition of `g` in the specification of `f`, this mechanism also makes specifications less readable, and leaks implementation details into the specification of `f`. We are currently experimenting with multiple strategies to cope with this, ranging from a better methodology and language support to structure specifications, to automatically inferring specs for some functions.

5 Conclusion

We described key aspects of the Move prover (MVP), a tool for formal verification of smart contracts written in the Move language. MVP has been successfully used to verify large parts of the Diem framework, and is used in continuous integration in production. The specification language is expressive, specifically by the powerful concept of global invariants. We described key implementation techniques which (as confirmed by our benchmarks) contributed to the scalability of MVP. One of the main areas of our future research is to improve specification productivity and reduce the effort of reading and writing specs, as well as to continue to improve speed and predictability of verification.

References

1. Astrauskas, V., Müller, P., Poli, F., Summers, A.J.: Leveraging rust types for modular specification and verification. PACMPL **3**(OOPSLA), 147:1–147:30 (2019)
2. Baranowski, M.S., He, S., Rakamaric, Z.: Verifying rust programs with SMACK. In: ATVA. Lecture Notes in Computer Science, vol. 11138, pp. 528–535. Springer (2018)

3. Barnett, M., Chang, B.Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: International Symposium on Formal Methods for Components and Objects. pp. 364–387. Springer (2005)
4. Barnett, M., DeLine, R., Fähndrich, M., Jacobs, B., Leino, K.R.M., Schulte, W., Venter, H.: The Spec# Programming System: Challenges and Directions, pp. 144–152. Springer Berlin Heidelberg, Berlin, Heidelberg (2008), https://doi.org/10.1007/978-3-540-69149-5_16
5. Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Gollamudi, A., Gonthier, G., Kobeissi, N., Kulatova, N., Rastogi, A., Sibut-Pinote, T., Swamy, N., Béguelin, S.Z.: Formal verification of smart contracts: Short paper. In: PLAS@CCS. pp. 91–96. ACM (2016)
6. Blackshear, S., Cheng, E., Dill, D.L., Gao, V., Maurer, B., Nowacki, T., Pott, A., Qadeer, S., Rain, Russi, D., Sezer, S., Zakian, T., Zho, R.: Move: A Language With Programmable Resources (2019), https://developers.libra.org/docs/move-paper
7. Blackshear, S., Nowacki, T., Qadeer, S., Mitchell, J.: The move borrow checker (2021), TBD
8. Clarke, D., Östlund, J., Sergey, I., Wrigstad, T.: Ownership types: A survey. In: Clarke, D., Noble, J., Wrigstad, T. (eds.) Aliasing in Object-Oriented Programming. Types, Analysis and Verification, Lecture Notes in Computer Science, vol. 7850, pp. 15–58. Springer (2013). https://doi.org/10.1007/978-3-642-36946-9_3, https://doi.org/10.1007/978-3-642-36946-9_3
9. ConsenSys: Mythril Classic: Security analysis tool for Ethereum smart contracts, https://github.com/skylightcyber/mythril-classic
10. Fatima Samreen, N., Alalfi, M.H.: Reentrancy vulnerability identification in ethereum smart contracts. In: 2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE). pp. 22–29 (2020)
11. Grishchenko, I., Maffei, M., Schneidewind, C.: A semantic framework for the security analysis of ethereum smart contracts. In: POST. Lecture Notes in Computer Science, vol. 10804, pp. 243–269. Springer (2018)
12. Hajdu, Á., Jovanovic, D.: solc-verify: A modular verifier for solidity smart contracts. CoRR **abs/1907.04262** (2019)
13. Hajdu, Á., Jovanovic, D.: SMT-Friendly Formalization of the Solidity Memory Model. In: ESOP. Lecture Notes in Computer Science, vol. 12075, pp. 224–250. Springer (2020)
14. Hildenbrandt, E., Saxena, M., Rodrigues, N., Zhu, X., Daian, P., Guth, D., Moore, B.M., Park, D., Zhang, Y., Stefanescu, A., Rosu, G.: KEVM: A complete formal semantics of the ethereum virtual machine. In: CSF. pp. 204–217. IEEE Computer Society (2018)
15. Lahiri, S.K., Chen, S., Wang, Y., Dillig, I.: Formal specification and verification of smart contracts for azure blockchain. CoRR **abs/1812.08829** (2018)
16. Leino, K.M.: Accessible software verification with dafny. IEEE Software **34**(06), 94–97 (nov 2017). https://doi.org/10.1109/MS.2017.4121212
17. Leino, R., Pit-Claudel, C.: Trigger selection strategies to stabilize program verifiers (January 2016), https://www.microsoft.com/en-us/research/publication/trigger-selection-strategies-stabilize-program-verifiers/
18. Lindner, M., Aparicius, J., Lindgren, P.: No panic! verification of rust programs by symbolic execution. In: INDIN. pp. 108–114. IEEE (2018)
19. Liu, J., Liu, Z.: A survey on security verification of blockchain smart contracts. IEEE Access **7**, 77894–77904 (2019)
20. Luu, L., Chu, D., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: ACM Conference on Computer and Communications Security. pp. 254–269. ACM (2016)
21. Maillard, K., Ahman, D., Atkey, R., Martínez, G., Hritcu, C., Rivas, E., Tanter, E.: Dijkstra monads for all. In: 24th ACM SIGPLAN International Conference on Functional Programming (ICFP) (2019), https://arxiv.org/abs/1903.01237

22. Matsakis, N.D., Klock, II, F.S.: The rust language. Ada Lett. **34**(3), 103–104 (Oct 2014). https://doi.org/10.1145/2692956.2663188, http://doi.acm.org/10.1145/2692956.2663188
23. Meyer, B.: Applying "design by contract". Computer **25**(10), 40–51 (Oct 1992). https://doi.org/10.1109/2.161279, https://doi.org/10.1109/2.161279
24. Miller, A., Cai, Z., Jha, S.: Smart contracts and opportunities for formal methods. In: International Symposium on Leveraging Applications of Formal Methods. pp. 280–299. Springer (2018)
25. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: TACAS. Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer (2008)
26. Nikolic, I., Kolluri, A., Sergey, I., Saxena, P., Hobor, A.: Finding the greedy, prodigal, and suicidal contracts at scale. In: ACSAC. pp. 653–663. ACM (2018)
27. Rosu, G., Serbanuta, T.: An overview of the K semantic framework. J. Log. Algebr. Program. **79**(6), 397–434 (2010)
28. The Coq development team: The coq proof assistant reference manual version 8.9 (2019), https://coq.inria.fr/distrib/current/refman/
29. The CVC Team: CVC5, https://github.com/cvc5/cvc5
30. The Diem Association: An Introduction to Diem (2019), https://libra.org/en-US/white-paper/
31. The Diem Association: The Diem Framework (2020), https://github.com/libra/libra/tree/master/language/stdlib
32. The Move Team: The Move Language Definition (2020), TBD
33. The Move Team: The Move Specification Language (2020), TBD
34. Tolmach, P., Li, Y., Lin, S., Liu, Y., Li, Z.: A survey of smart contract formal specification and verification. CoRR **abs/2008.02712** (2020), https://arxiv.org/abs/2008.02712
35. Toman, J., Pernsteiner, S., Torlak, E.: Crust: A bounded verifier for rust (N). In: ASE. pp. 75–80. IEEE Computer Society (2015)
36. Tsankov, P., Dan, A.M., Drachsler-Cohen, D., Gervais, A., Bünzli, F., Vechev, M.T.: Securify: Practical security analysis of smart contracts. In: ACM Conference on Computer and Communications Security. pp. 67–82. ACM (2018)
37. Yang, Z., Lei, H.: Formal process virtual machine for smart contracts verification. CoRR **abs/1805.00808** (2018)
38. Yang, Z., Lei, H.: Fether: An extensible definitional interpreter for smart-contract verifications in coq. IEEE Access **7**, 37770–37791 (2019)
39. Zhong, J.E., Cheang, K., Qadeer, S., Grieskamp, W., Blackshear, S., Park, J., Zohar, Y., Barrett, C., Dill, D.L.: The Move Prover. In: Lahiri, S.K., Wang, C. (eds.) Computer Aided Verification. pp. 137–150. Springer International Publishing (2020)

Fig. 8: Requires, Ensures, and AbortsIf Injection

```
1   fun f(x: u64, y: u64): u64 { x + y }
2   spec f {
3     requires x < y;
4     aborts_if x + y > MAX_U64;
5     ensures result == x + y;
6   }
7   fun g(x: u64): u64 { f(x, x + 1) }
8   spec g {
9     ensures result > x;
10  }
11  ⇝
12  fun f(x: u64, y: u64): u64 {
13    spec assume x < y;
14    let result = x + y;
15    spec assert result == x + y;      // ensures of f
16    spec assert                       // negated abort_if of f
17      !(x + y > MAX_U64);
18    result
19  } onabort {
20    spec assert                       // abort_if of f
21      x + y > MAX_U64;
22  }
23  fun g(x: u64): u64 {
24    spec assert x < x + 1;            // requires of f
25  if inlined
26    let result = inline f(x, x + 1);
27  elif opaque
28    if (x + x + 1 > MAX_U64) abort;   // aborts_if of f
29    spec assume result == x + x + 1; // ensures of f
30  endif
31    spec assert result > x;           // ensures of g
32    result
33  }
```

A Injection of Function Specifications

During function specification injection, Move specifications are reduced to basic assume/assert statements added to the Move code. Those statements represent instructions to the solver backend about what propositions can be assumed and which need to be asserted (verified) at a given program point. In this appendix, we cover how *function specification conditions* are injected.

A.1 Pre- and Post conditions

The injection of basic function specifications is illustrated in Fig. 8. An extension of the Move source language is used to specify abort behavior. With `fun f(){ .. } onabort { conditions }` a Move function is defined where `conditions` are assume or assert statements that are evaluated at every program point the function aborts (either implicitly or with an `abort` statement). This construct simplifies the presentation and corresponds to a per-function abort block on bytecode level which is target of branching.

An aborts condition is translated into two different asserts: one where the function aborts and the condition must hold (line 21), and one where it returns and

Fig. 9: Modifies Injection

```
1   fun f(addr: address) { move_to<T>(addr, T{}) }
2   spec f {
3     pragma opaque;
4     ensures exists<T>(addr);
5     modifies global<T>(addr);
6   }
7   fun g() { f(0x1) }
8   spec g {
9     modifies global<T>(0x1); modifies global<T>(0x2);
10  }
11  ⇝
12  fun f(addr: address) {
13    let can_modify_T = {addr};      // modifies of f
14    spec assert addr in can_modify; // permission check
15    move_to<T>(addr, T{});
16  }
17  fun g() {
18    let can_modify_T = {0x1, 0x2};  // modifies of g
19    spec assert {0x1} <= can_modify_T; // permission check
20    spec havoc global<T>(0x1);      // havoc modified memory
21    spec assume exists<T>(0x1);     // ensures of f
22  }
```

the condition must *not* hold (line 17). If there are multiple `aborts_if`, they are ored. If there is no abort condition, no asserts are generated. This means that once a user specifies aborts conditions, they must completely cover the abort behavior of the code. (The prover also provides an option to relax this behavior, where aborts conditions can be partial and are only enforced on function return.)

For a function call site we distinguish two variants: the call is *inlined* (line 25) or it is *opaque* (line 27). For inlined calls, the function definition, with all injected assumptions and assertions turned into assumptions (as those are considered proven) is substituted. For opaque functions the specification conditions are inserted as assumptions. Methodologically, opaque functions need precise specifications relative to a particular objective, where as in the case of inlined functions the code is still the source of truth and specifications can be partial or omitted. However, inlining does not scale arbitrarily, and can be only used for small function systems.

Notice we have not discussed the way how to deal with relating pre and post states yet, which requires taking snapshots of state (e.g. `ensures x == old(x)+ 1`); the example in Fig. 8 does not need it. Snapshots of state will be discussed for global update invariants in Sec. 3.2.

Modifies The `modifies` condition specifies that a function only changes specific memory. It comes in the form `modifies global<T>(addr)`, and its injection is illustrated in Fig. 9.

A type check is used to ensure that if a function has one or more `modifies` conditions all called functions which are *opaque* have a matching modifies declaration. This is important so we can relate the callees memory modifications to that what is allowed at caller side.

At verification time, when an operation is performed which modifies memory, an assertion is emitted that modification is allowed (e.g. line 14). The permitted ad-

Fig. 10: Emits Injection

```
1   use Std::Event;
2   struct E has drop, store { m: u64 }
3   fun f(h: &mut Event::EventHandle<E>, x: u64) {
4     Event::emit_event(h, E{m:0});
5     if (x > 0) {
6       Event::emit_event(h, E{m:x});
7     }
8   }
9   spec f {
10    emits E{m:0} to h;
11    emits E{m:x} to h if x > 0;
12  }
13  ⇝
14  fun f(h: &mut Event::EventHandle<E>, x: u64) {
15    es = Mvp::ExtendEventStore(es, h, E{m:0}); // emitting event
16    if (x > 0) {
17      es = Mvp::ExtendEventStore(es, h, E{m:x}); // emitting event
18    }
19    let actual_es = Mvp::subtract(es, old(es)); // events emitted by f
20    let expected_es = Mvp::CondExtendEventStore( // specified events
21      Mvp::ExtendEventStore(Mvp::EmptyEventStore, E{m:x}, h),
22      E{m:x}, h, x>0);
23    spec assert Mvp::includes(expected_es, actual_es); // spec completeness
24    spec assert Mvp::includes(actual_es, expected_es); // spec relevance
25  }
```

dresses derived from the modifies clause are stored in a set `can_modify_T` generated by the transformation. Instructions which modify memory are either primitives (like `move_to` in the example) or function calls. If the function call is inlined, modifies injection proceeds (conceptually) with the inlined body. For opaque function calls, the static analysis has ensured that the target has a modifies clause. This clause is used to derive the modified memory, which must be a subset of the modified memory of the caller (line 19).

For opaque calls, we also need to *havoc* the memory they modify (line 20), by which is meant assigning an unconstrained value to it. If present, `ensures` from the called function, injected as subsequent assumptions, are further constraining the modified memory.

Emits The injection for the `emits` clause is illustrated in Fig. 10. The `emits` clause specifies the events that a function is expected to emit. It comes in the form `emits message to handle if condition` (e.g., line 11). The condition part (i.e., `if condition`) can be omitted if the event is expected to be emitted unconditionally (e.g., line 10).

The function call to `Event::emit_event` (e.g., line 4) is transformed into the statement to extend `es` with the event to emit (e.g., line 15). `es` is a global variable of type `EventStore` which is a map where the key is an event handle and the value is the event stream of the handle (modeled as a bag of messages).

In line 19, `actual_es` represents the portion of the EventStore that only comprises the events that the program (i.e., `f`) actually emits. In line 20, `expected_es` is constructed from the `emits` specification which contains all of the expected

Fig. 11: Data Invariant Injection

```
1   struct S { a: u64, b: u64 }
2   spec S { invariant a < b }
3   fun f(s: S): S {
4     let r = &mut s;
5     r.a = r.a + 1;
6     r.b = r.b + 1;
7     s
8   }
9   ~~>
10  fun f(s: S): S {
11    spec assume s.a < s.b;      // assume invariant for s
12    let r = Mvp::local(s, F_s); // begin mutation of s
13    r = Mvp::set(r, Mvp::get(r)[a = Mvp::get(r).a + 1]);
14    r = Mvp::set(r, Mvp::get(r)[b = Mvp::get(r).b + 1]);
15    spec assert                 // invariant enforced
16      Mvp::get(r).a < Mvp::get(r).b;
17    s = Mvp::get(r);            // write back to s
18    s
19  }
```

events specified by the `emits` clauses. Having these, two assertions using `Mvp::includes` (multiset inclusion relation per event handle) are injected. One asserts that `expected_es` includes `actual_es`, meaning that the function only emits the events that are expected (e.g., line 23). This would be violated if there is any event emitted by `f` that is not covered by some `emits` clause. Another asserts that `actual_es` includes `expected_es`, meaning that the function emits all of the events that are expected (e.g., line 24). This would be violated if `f` does not emit the expected event which a `emits` clause specifies.

We also handle opaque calls properly although it is not illustrated in Fig. 10. Suppose `f` is an opaque function, and another function `g` calls `f`. In the transformation of `g`, the event store `es` extends with the expected events of `f` (i.e., the events specified by the `emits` clauses of `f`) in a similar way to how `expected_es` is constructed in line 20.

A.2 Data Invariants

A data invariant specifies a constraint over a struct value. The value is guaranteed to satisfy this constraint at any time. Thus, when a value is constructed, the data invariant needs to be verified, and when it is consumed, it can be assumed to hold.

In Move's reference semantics, construction of struct values is often done via a sequence of mutations via mutable references. It is desirable that *during* such mutations, assertion of the data invariant is suspended. This allows to state invariants which reference multiple fields, where the fields are updated step-by-step. Move's borrow semantics and concept of mutations provides a natural way how to defer invariant evaluation: at the point a mutable reference is released, mutation ends, and the data invariant can be enforced. In other specification formalisms, we would need a special language construct for invariant suspension. Fig. 11 gives an example, and shows how data invariants are reduced to assert/assume statements.

The implementation of data invariants hooks into the reference elimination (Sec. 3.1). As part of this the lifetime of references is computed. Whenever a reference is re-

leased and the mutated value is written back, we also enforce the data invariant. In addition, the data invariant is enforced when a struct value is directly constructed.

B Corrected Account Example

To fix the verification errors from the account example in Fig. 1 and Fig. 2, the following changes would need to be made:

```
module Account {
  ...

  fun withdraw(account: address, amount: u64) acquires Account {
    assert(amount <= AccountLimits::max_decrease(), Errors::
        invalid_argument()); // MISSING
    let balance = &mut borrow_global_mut<Account>(account).balance;
    assert(*balance >= amount, Errors::limit_exceeded());
    assert(*balance - amount >= AccountLimits::min_balance(), Errors
        ::invalid_argument()); // MISSING
    *balance = *balance - amount;
  }

  spec transfer {
    ...
    aborts_if !exists<Account>(from_addr); // MISSING
    aborts_if !exists<Account>(to); // MISSING
    aborts_if amount > AccountLimits::max_decrease(); // MISSING
    aborts_if bal(from_addr) - amount < AccountLimits::min_balance();
        // MISSING
  }
}
```