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

1 Introduction

Smart contracts have been for long considered a promising domain to employ formal verification. On the one hand, smart contracts deal with computations which require highest correctness assurance, as they may manipulate assets of large value with many rules and regulations in place. On the other hand, smart contracts have multiple properties (sandboxed, transactional, bounded computation steps) which make them technically well-suited for formal verification.

In this paper we report about the *Move prover* (abbreivated MVP), a tool for the *Move* smart contract language [1], which has been developed together with the Diem blockchain [8]. The Move language supports specifying pre- and post-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 these theoretical restrictions, MVP is capable of verifying the full Diem framework [9], which is the Move implementation of the Diem blockchain [8]. 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, typically in seconds*, which can be seen as a significant practical result for formal verification adoption.

From the point of the first Move Prover publication in [13], many improvements have been made to make such usage practical. 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 by an order of magnitude, 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 [4], the current version only rarely runs in 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.

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) {
    withdraw(Signer::address_of(from), amount);
    deposit(to, amount);
  }
}
```

2 Move and the Prover

Move was developed for the Diem blockchain [8], 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. 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

Fig. 2: Account Example Specification

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

where structs are defined. An illustration of a Move contract is given in Fig. 1 (for a more complete description see the Move Book [10]). 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 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* [5]. 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 [11]). This adds specifications for the `transfer` function, a helper function `bal` used 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 underflow or overflow.

We have omitted specifications for the `withdraw` and `deposit` functions intentionally. MVP supports keeping specs for non-recursive functions open, 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 as 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 the prover on the program and specification of `module Account` produces multiple errors. The first is this one:

TODO(wrwg): update example

```
error: abort not covered by any of the ‘aborts_if‘ clauses

    --- account.move:15:3 ---
    |
 15 |  public fun withdraw(account: address, amount: u64): Coin
    .
 18 |          &mut borrow_global_mut<Account>(account).balance;
    .                ----------------- abort happened here
    |
    =     at account.move:15:3: withdraw
    =         account = 0x19, amount = 15724
    =     at account.move:18:14: withdraw (ABORTED)
```

The prover has 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, the Move prover 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:

```
error: global memory invariant does not hold

    --- account.move:43:5 ---
    |
 43 | invariant forall acc: address where exists<Account>(acc):
 44 |     bal(acc) >= AccountLimits::MIN_BALANCE;
    |
    .
    =     at account.move:21:35: withdraw

error: global memory invariant does not hold

    --- account.move:45:5 ---
```

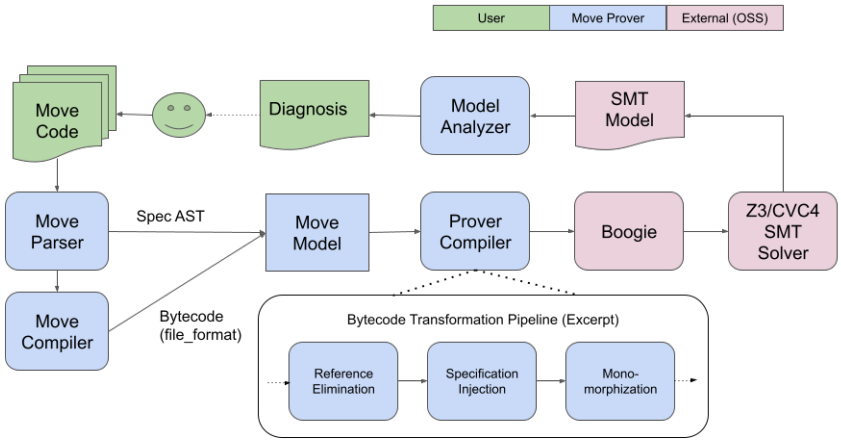

Fig. 3: Move Prover Architecture

```
   |
45 | invariant update
46 |   forall acc: address where exists<Account>(acc):
47 |     old(bal(acc)) - bal(acc) <= AccountLimits::MAX_DECREASE;
   .
   =     at account.move:21:35: withdraw
```

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 how to fix this problem, which would require to add some more `assert` statements to the code and abort if the limits are not met.

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 metdata 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 [6]. 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 [12] and CVC5 [7]; the default choice for the Move prover is currently Z3.

When the SMT solver produces a `sat` or `unknown` result (of the negation of the verification condition Boogie generates), it produces a model witness. The Move Prover attempts to translate this model back into a diagnostic which a user can associate with the original Move code (as has been illustrated in Sec. 2.) For example, execution traces leading to the verification failure are shown, with assignments to variables used in this trace, extracted from the model. Also the Move Model will be consulted to retrieve the original source information and display it with the diagnosis.

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* [3] 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 [2]):

- 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 existance 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 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 replace it with a sub-reference (`r' = &mut r.f`). The mutation ends when `r` (or 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 }
```

While the setup in this 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. This is discussed in App. A.

3.2 Global Invariant Injection

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, as in `invariant forall a: address: P(global<S>(a))`. 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 global 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.

Proof Methodology Inductive invariants are proved by induction over the evolution of the global memory. The base case is that the invariant must hold in the empty state that preceeds 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.

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())` satisifies 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 proved. For each invariant, there is a defined subset of Move modules (called a *cluster*). If the invariant is proved 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 proved. The cluster must contain every function that can invalidate the invariant, and, in case of invariant suspension, all callers of such a function. Moreover, it must contain any function which calls a friend function in a module for which the invariant is suspended. 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. 4 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 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,

Fig. 4: 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];
}
```

Fig. 5: 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];
}
```

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. For soundness, a property must hold for every possible instantiation of type parameters. So, rather than checking whether some of the types mentioned in the invariant are equal to some of the types accessed or modified by a function MVP needs to discover whether there is any possible instantiation of type parameters that might allow the instantiated function to invalidate an instantiated invariant. In other words, it needs to know whether the each type in the invariant can be unified with a type accessed or modified by

Fig. 6: 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) }
```

the function. Consider the example in Fig. 5. 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*. Like with genericity in most modern program languages, this is possible in Move because the number of instantiations is statically known for a given program fragment.

Basic Monomorphization To verify a generic function, monomorphization skolemizes the type parameter into a given type. It then, for all functions which are inlined, inserts their code specializing it for the given type instantiation, including specialization of all used types. Fig. 6 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.

- Foreach 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 instaniation 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. 5. 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

TODO(wrwg): complete

- Benchmarks
- Open Problems
- Related Work
- ...

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 three 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. 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
2. Blackshear, S., Nowacki, T., Qadeer, S., Mitchell, J.: The move borrow checker (2021), TBD
3. 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
4. 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/
5. Meyer, B.: Applying "design by contract". Computer **25**(10), 40âĂŞ51 (Oct 1992). https://doi.org/10.1109/2.161279, https://doi.org/10.1109/2.161279
6. The Boogie Team: Boogie Intermediate Verification Language, https://github.com/boogie-org/boogie
7. The CVC Team: CVC5, https://github.com/cvc5/cvc5
8. The Diem Association: An Introduction to Diem (2019), https://libra.org/en-US/white-paper/
9. The Diem Association: The Diem Framework (2020), https://github.com/libra/libra/tree/master/language/stdlib
10. The Move Team: The Move Language Definition (2020), TBD
11. The Move Team: The Move Specification Language (2020), TBD

12. The Z3 Team: Z3 Prover, https://github.com/Z3Prover/z3
13. 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)

A Elimination of Mutable References with Dynamic Value

In Sec. 3.1 we described how mutable references are eliminated in the case their root location is statically known. This covers the majority of typical Move code. However, there are also important cases where a root location depends on runtime decisions:

```
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. 7 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.
- Similarily, `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.
- With `r' = Mvp::field(r, FIELD_ID)` a mutation value for a subreference 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. The predicate `Mvp::is_field(r, FIELD_ID)` tests whether it is derived from the given field.

Implementation 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.

The transformation 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

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

Fig. 7: Elimination of Mutable References

```
1   fun increment(x: &mut u64) { *x = *x + 1 }
2   fun increment_field(s: &mut S) {
3     let r = if (s.f > 0) &mut s.f else &mut s.g;
4     increment(r)
5   }
6   fun caller(p: bool): (S, S) {
7     let s1 = S{f:0, g:0}; let s2 = S{f:1, g:1};
8     let r = if (p) &mut s1 else &mut s2;
9     increment_field(r);
10    (s1, s2)
11  }
12  ⇝
13  fun increment(x: Mut<u64>): Mut<u64> { Mvp::set(x, Mvp::get(x) + 1) }
14  fun increment_field(s: Mut<S>): Mut<S> {
15    let r = if (s.f > 0) Mvp::field(s.f, S_F) else Mvp::field(s.g, S_G);
16    r = increment(r);
17    if (Mvp::is_field(r, S_F))
18      s = Mvp::set(s, Mvp::get(s)[f = Mvp::get(r)]);
19    if (Mvp::is_field(r, S_G))
20      s = Mvp::set(s, Mvp::get(s)[g = Mvp::get(r)]);
21    s
22  }
23  fun caller(p: bool): S {
24    let s1 = S{f:0, g:0}; let s2 = S{f:1, g:1};
25    let r = if (p) Mvp::mklocal(s1, CALLER_s1)
26             else Mvp::mklocal(s2, CALLER_s2);
27    r = increment_field(r);
28    if (Mvp::is_local(r, CALLER_s1))
29      s1 = Mvp::get(r);
30    if (Mvp::is_local(r, CALLER_s2))
31      s2 = Mvp::get(r);
32    (s1, s2)
33  }
```

they relate to each other: e.g. `r' = &mut r.f` creates a edge from `r` to `r'` labelled 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 7. 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. 7). The presence of multiple possible origins leads to case distinctions via `Mvp::is_X` predicates; however, these cases are rare in actual Move programs.

B 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.

B.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 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.

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 of
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  }
```

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 addresses 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.

Fig. 9: Modifies Injection

```
fun f(addr: address) { move_to<T>(addr, T{}) }
spec f {
  pragma opaque;
  ensures exists<T>(addr);
  modifies global<T>(addr);
}
fun g() { f(0x1) }
spec g {
  modifies global<T>(0x1); modifies global<T>(0x2);
}
⇝
fun f(addr: address) {
  let can_modify_T = {addr};      // modifies of f
  spec assert addr in can_modify; // permission check
  move_to<T>(addr, T{});
}
fun g() {
  let can_modify_T = {0x1, 0x2};  // modifies of g
  spec assert {0x1} <= can_modify_T; // permission check
  spec havoc global<T>(0x1);      // havoc modified memory
  spec assume exists<T>(0x1);     // ensures of f
}
```

B.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. 10 gives an example, and shows how data invariants are reduced to assert/assume statements.

Implementation The implementation hooks into the reference elimination (Sec. 3.1). As part of this the lifetime of references is computed. Whenever a reference is released 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.

Fig. 10: Data Invariant Injection

```
struct S { a: u64, b: u64 }
spec S { invariant a < b }
fun f(s: S): S {
  let r = &mut s;
  r.a = r.a + 1;
  r.b = r.b + 1;
  s
}
⇝
fun f(s: S): S {
  spec assume s.a < s.b;      // assume invariant for s
  let r = Mvp::local(s, F_s); // begin mutation of s
  r = Mvp::set(r, Mvp::get(r)[a = Mvp::get(r).a + 1]);
  r = Mvp::set(r, Mvp::get(r)[b = Mvp::get(r).b + 1]);
  spec assert                 // invariant enforced
    Mvp::get(r).a < Mvp::get(r).b;
  s = Mvp::get(r);            // write back to s
  s
}
```