Towards Practical Formal Verification of Smart Contracts

– Technical Report –

DRAFT

David Dill Wolfgang Grieskamp Junkil Park
Shaz Qadeer Meng Xu Emma Zhong

July 30, 2021

Contents

1 Introduction

Move [1] is a new high-level programming language for writing smart contracts. Move and it's underlying engine, the Move Virtual Machine, have been designed with *formal verification* in mind. Specification related language support is integrated into the Move language, and the *Move prover* tool has been developed, allowing to verify specifications against the implementation.

In this paper, we describe for the first time the methodology and theory behind *practical formal verification* in Move. Since the earlier publication about the Move prover tool in [14], many changes have been made to the Move specification language and the prover implementation. Those changes went hand-in-hand with the evolution of the *Diem framework* [9], which is a Move library for smart contracts running on 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*, being integrated into the regular development process of Diem.

The cornerstones for the success of this project are seen in the following aspects. First, the Move language was designed with verification in mind, making it amenable for mechanized proof. Specifically its memory safety properties and borrow semantics contribute to this. Second, the fully sandboxed execution model of Move reduces the problem of outcalls to unspecified, open-ended code which many verification approaches face. Third, the state of the art in SMT tools like Z3 [12] give rise to powerful practical applications. Last but not least, the co-development of the Diem framework with the specification language and prover provided a valuable feedback loop for continuous improvement.

While those results look promising, constituting one of the larger recent applications of formal verification in industry, the technology is not yet fully ready for mainstream usage. A major classical obstacle of SMT-based verification remains: as we are dealing with undecidable problems, heuristics in the solver can fail, leading to occasional verification timeouts, which require support of specialized engineers to solve. Also, specifications are arguably harder to write than code, and require significant effort. Moreover, diagnosis produced by the prover on verification failures, while already better than one would expect, need to be further improved. We describe the obstacles for mainstream usability, and our ideas how they might be overcome in the conclusion of this paper.

Acknowledgement Many more people have contributed to the Move Prover: Sam Blackshear, Mathieu Baudet, Todd Nowacki, Bob Wilson, Tim Zaikan,

…(list interns and other Move team members)

2 Overview of Move

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 state*. Updates are executed in a *transactional* style: the next state they compute will only be committed if their computation has finished successfully and the result can be merged back into the global state. Semantically, a Move execution can therefore be interpreted as a labelled transition system (i.e. interleaved execution steps). Any state evolving system which is adequately modeled by this semantics, not just blockchains, can be programmed in Move (for example, transactions on a concurrent data base, or training steps in an iterative ML algorithm).

The Move language allows to define global state in terms of so-called *resources*. Resources are data structures which are stored in *persistent* global memory indexed by *account addresses*. For example, the Move expression **exists**`<Balance>(account)` determines whether the resource `Balance` exists at address `account`. As resource types can be generic (for example, `Balance<Currency>`), an index for a resource is a tuple of types and the account address: semantically, for each resource type R there is a partial memory function $\mathcal{M}_R \in \overline{\mathcal{T}} \times \mathcal{A} \rightarrow R_\perp$, with $\overline{\mathcal{T}}$ a sequence of types use to instantiate $R\langle\overline{\mathcal{T}}\rangle$, and $\mathcal{A}$ the domain of addresses. 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 state $\mathcal{M}$. The execution of this function can fail via a well-defined abort mechanism, in which case $\mathcal{M}$ stays unmodified. An environment emits a sequence of calls to such scripts, thereby evolving $\mathcal{M}$. To understand this execution as an LTS, consider the set of states to be $\mathcal{M}$, the labels the names of transaction scripts combined with a set of concrete parameters, and the transition relation defined by the transaction scripts. Abortion of the transaction function creates a label with the script name, the parameters, and information about the abort reason, which cycles on the current state.

A transaction script is written in Move as an imperative function which can read and write the global memory $\mathcal{M}$. Move uses a specific style of imperative programming based on *borrow semantics* [3], as popularized in the programming language Rust [4, 13]. For the verification problem borrow semantics is very important. While allowing references into structured data, those are guaranteed to be safe by the *borrow checker* [2], which is run during bytecode loading time, and which verification can assume. Furthermore, the notorious hard verification problem of aliasing of references in the presence of mutation is eliminated. Mutation always starts from a root location either in global memory or on the execution stack, and while a tree starting from this root is mutated, no other access can happen anywhere in the tree. Intuitively, borrow semantics allows to move a mutation 'cursor' down the tree, which follows linear typing discipline. Because of this property, mutable reference parameters to functions can be converted to input/output parameters, and verification of Move can avoid the traditionally hard problems caused by aliasing of mutable references.

2.1 Programming in Move

In Move, one defines transactions via top-level 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 resource types and other structured data is defined.

We illustrate the language by example (for a more complete description of Move, see the online documentation [10]). The example is a simple account which holds a balance of coins and is given in Fig. 1. The transaction is to transfer coins from one to another account [1].

- The struct `Account::Coin` represents units of currency. The **has store** ability of the `Coin` struct indicates that it can be stored as a field in another struct. Notice that by default, in Move, structs have *linear* semantics: a `Coin` cannot be copied and dropped without explict destruction (as on line 19). This is useful to prevent accidental duplication or lost of coins. To indicate that the struct can be copied and dropped, one would need to add the abilities **has copy, drop**.

- `Coin` is aggregated in the struct `Account` for representing a *balance*; the ability **has key** indicates that this struct can be stored as a resource in global memory.

- The `Account::withdraw` function subtracts a value from the balance, returning a new `Coin` for the withdrawn amount. It uses the builtin function **borrow_global_mut<T>(address)** which returns a mutable reference to the `Account` resource. Similarily, `Account::deposit` takes a coin which is destructed and its amount added to the account.

- The `acquire Account` modifier on a function declaration indicates that the function will borrow the `Account` global memory as a whole – i.e. for every account address. The Move borrow checker will reject a call to such functions if any account resources are already borrowed, implementing memory safety for Move [2].

- The **assert** statement causes a Move transaction to abort execution if the condition is not met, with the specified error code. No effects on the global state occur on abort. Abortion can also happen implicitly; for example,

[1] Indeed, for a complete system, transactions like creating an account and funding it would be needed, but we leave this aspect out here.

Figure 1: Account Example Program

```
1  module Account {
2    struct Coin has store {
3      value: u64
4    }
5    struct Account has key {
6      balance: Coin,
7    }
8
9    public fun withdraw(account: address, amount: u64): Coin
10   acquires Account {
11     let balance = &mut borrow_global_mut<Account>(account).balance;
12     assert(balance.value >= amount, Errors::limit_exceeded());
13     balance.value = balance.value - amount;
14     Coin{value: amount}
15   }
16
17   public fun deposit(account: address, check: Coin)
18   acquires Account {
19     let Coin{value: amount} = check; // Consume coin
20     let balance = &mut borrow_global_mut<Account>(account).balance;
21     assert(balance.value <= Limits::max_u64() - amount,
22            Errors::limit_exceeded());
23     balance.value = balance.value + amount;
24   }
25
26   public(script) fun transfer(from: &signer, to: address, amount: u64) {
27     let coin = Account::withdraw(Signer::address_of(from), amount);
28     Account::deposit(to, move(coin))
29   }
30 }
```

the expression **borrow_global_mut**<T>(addr) will abort if no resource T exists at addr.

- The script Account::transfer is a top-level entry point into this Move program, calling Account::withdraw and Account::deposit. The call to the builtin function **move** at line 28 illustrates how the linear coin value travels from one call to another.

- 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 transaction.

2.2 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 invari-

Figure 2: Account Example Specification

```
1 module Account {
2   spec withdraw {
3     aborts_if bal(account) < amount;
4     ensures bal(account) == old(bal(account)) - amount;
5     ensures result == Coin{value: amount};
6     modifies global<Account>(acc);
7   }
8 
9   spec deposit {
10    aborts_if bal(account) + check.value > Limits::max_u64();
11    ensures bal(account) == old(bal(account)) + check.value;
12    modifies global<Account>(acc);
13  }
14
15  spec fun bal(acc: address): num {
16    global<Account>(acc).balance.value
17  }
18
19  invariant forall acc: address where exists<Account>(acc):
20    bal(acc) >= AccountLimits::MIN_BALANCE;
21
22  invariant update forall acc: address where exists<Account>(acc):
23    old(bal(acc)) - bal(acc) <= AccountLimits::MAX_DECREASE;
24 }
```

ants 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]).

- The function specification blocks **spec** `withdraw` and **spec** `desposit` specify when those functions abort, the expected effect on the global memory, and its return value (the return value is represented by the well-known name `result`).

- As common in this style of specifications, in the **ensures** statement, by default the post-state of the function is referred to, whereas the form **old**`(..)` can be used to access the pre-state.

- We are using the helper function `bal(`**address**`)` defined on line 15 to access the value of the account balance. Helper functions can access state and can be transparently used within **old**`(..)`; the function is then evaluated in the pre-state.

- The **modifies** statement on line 6 specifies that this function only changes the indicated memory but no other memory.
- The specification contains two invariants over global memory. The first invariant on line 19 states that a balance can never drop underneath a certain minimum. The second invariant on line 22 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.

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.

The constructs we have seen so far are only a subset of the available features of the Move specification language. Notably, the language supports the following additional features:

- Function preconditions via the **requires**-clause.
- Data invariants for **struct** types, as a predicate over the field values.
- Means to abstract commonly used specification fragments in so-called *specification schemas* which can then be included in other specification blocks.

2.3 Running the Prover

The Move prover is a tool which supports verification of specifications as shown above. The prover 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, as mentioned. The first is this one:

TODO(wrwg): make line number symbolic so they align with figures

```
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 counter example found by the prover 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 ---
    |
 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 Implementation

In this section, an overview of the Move Prover implementation will be provided. The formal content of the discussion is kept lightweight; a formalization of some aspects is given in appendices.

3.1 Basic Architecture

The architecture of the Move Prover is illustrated in Fig. 3. Move code (consisting of Move programs and specifications) is given as input to the Move tool chain, which produces two artifacts: the abstract syntax tree (AST) of the specifications in the code, as well as the translated Move bytecode for the program part. It is essential that the Prover interprets the Move program on bytecode level, not on the intermediate AST: this way we verify the "source of truth" which is also executed in the Move VM. Only the specification parts are passed on as AST. The *Move Model* is a component which 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.

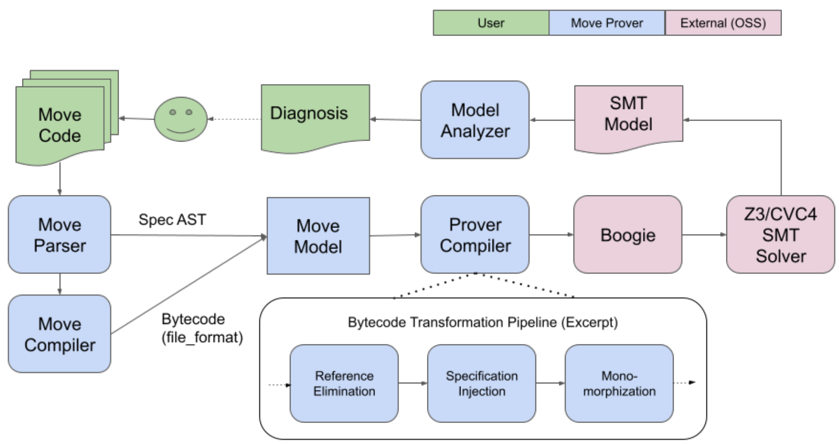

Figure 3: Move Prover Architecture

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 Move 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 undertakes some effort to translate this model back into diagnosis which a user can associate it with the original Move code (as has been illustrated in Sec. 2.3.) 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 as well as the encoding and translation to Boogie.

3.2 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.

3.2.1 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.

3.2.2 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. We can create a copy of the data in `l` and single-thread this 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`. We effectively turned 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 significant 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 uses cases in every day Move code, there are more complex ones to consider, namely that the value of a reference depends on runtime decisions:

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

Additional runtime information 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 from, so we can write back the updated value correctly. Fig. 3.2.2 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.

Figure 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)
}
```

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.[2] (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 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

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

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 3.2.2. 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. 3.2.2). The presence of multiple possible origins leads to case distinctions via `Mvp::is_X` predicates; however, these cases are rare in actual Move programs.

Performance **TODO(wrwg)**: We may want to identify some historical benchmarks before memory model.

3.3 Specification Injection

Move specifications are reduced to basic assume/assert statements injected into the Move code. As usual, an assume statement formulates a condition which can be assumed to hold for verification at a given program point, whereas an assert statement a condition which needs to be verified.

Specification instrumentation starts from a program for which references have been removed, and mutation is transformed into a read-update-write cycle (Sec. 3.2).

3.3.1 Modular Verification

Modular verification applies to all types of injections, and its principles are therefore described first. When the Move prover is run, it takes as input a set of Move modules which is closed under the transitive dependency relation (module imports). However, only a subset of those modules are *verification target* (typically just one module). It is assumed that the tool environment ensures that modules in the dependency relation which are not target of verification have already successfully verified.

From the set of target modules, the set of *target functions* is derived. This set might be enriched by additional functions which need verification because of global invariants, as discussed in Sec. 3.3.5. The resulting set of target functions will then be verified one-by-one, assuming that any called functions have successfully verified. If a called function is among the target functions, it might in fact not verify; however, in this case a verification error will be reported at the called function, and the verification result at the caller side can be ignored.

3.3.2 Pre- and Postconditions

The injection of basic function specifications is illustrated in Fig. 5. 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

Figure 5: 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 }
```

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 or-ed. If there is no aborts 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). In both cases, it is assumed that the called function is verified (see Modular Verification, Sec. 3.3.1). For inlined calls, the function definition, with all injected assumptions and assertions turned into assumptions (as

Figure 6: 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 move_to
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 call f
20   spec havoc global<T>(0x1);          // havoc memory modified by f
21   spec assume exists<T>(0x1);         // ensures of f
22 }
```

those are considered proven) is substituted. For opaque functions the specification conditions are inserted as assumptions. Methodlogically, opaque functions need precise specificatons 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. 5 does not need it. Snapshoting of state will be discussed for global update invariants in Sec. 3.3.5.

3.3.3 Modifies

The **modifies** condition specifies that a function only changes specific memory. It comes in the form **modifies global**`<T>(addr)`, and its onjection is illustrated in Fig. 6.

A type check is used to ensure that if a function specificies a modifies all called functions which are *opaque* declare modifies for the same type `T`. 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 addresses derived from the modifies clause are stored in a set `can_modify_T` generated by the transformation. Instructions which modify memory are either

Figure 7: 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 parameter
  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                 // end mutation: invariant enforced
    Mvp::get(r).a < Mvp::get(r).b;
  s = Mvp::get(r);            // write back to s
  s
}
```

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 unconstraint value to it. If present, **ensures** from the called function, injected as subsequent assumptions, are further constraining the modified memory.

3.3.4 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. 7 gives an example, and shows how data invariants are reduced to assert/assume statements.

Implementation The implementation hooks into the reference elimination (Sec. 3.2). 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

Figure 8: Basic Global 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];
}
```

data invariant. In addition, the data invariant is enforced when a struct value is directly constructed.

3.3.5 Global Invariants

Global invariants appear on Move module level and constraint the content of the memory. While the basic injection of global invariants is relative simple, they cause significant complexity with features like modular verification, suspension, and generics. We first discuss the basic model, then extend it stepwise.

Basic Global Invariants Fig. 8 contains an example for the supported invariant types and their injection into code. The first invariant, `I1`, is a regular state invariant. It is assumed on function entry, and asserted after the state update. The second, `I2`, is a state 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.

Notice that invariant injection can lead to inconsistencies. Consider the following code fragment:

```
invariant [I] forall a: address: global<S>(a).value > 0;
⇝
spec assume global<S>(0).value == 0;  // context, e.g. from a requires
spec assume I;                        // injected
```

We currently do not check whether an invariant is satisfiable before we assume it, but rather rely on a generic consistency checker for specifications.

The global invariant injection is optimized by knowledge of the prover, obtained by static analysis, about (transitively) accessed memory. For opaque functions (including also builtin functions) this information is obtained via the modifies clause. For other function it is determined from the code. The full

Figure 9: Genericity

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

story is more evolved because of the presence of both generic invariants and functions, and is discussed below.

Genericity In the case of generic invariants and functions, we must use *type unification* to determine which invariants are injected. Consider the example in Fig. 9. 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 successfull unification will result in a substitution for both. On successfull 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. 3.4.

Modular Verification

Disabling Invariants

Figure 10: Basic Monomorphization

```
fun f<T>(x: T) { g<S<T>>(S(x)) }
⇝
struct given_T{}
fun f_T(x: given_T) { g_S_T(S_T(x)) }
```

3.4 Monomorphization

Monomorphization is the process of removing all generics from a Move programs. It greatly improves the performance of the backend solvers.

3.4.1 Generic Functions

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. 10 sketches this approach.

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

3.4.2 Type Dependent Code

In Move, types are (almost) not able to influence runtime semantics. There is one exception: if memory is indexed by a generic, as in `S<T>`. One can essentially implement a generic type check in move:

```
fun init() { move_to<S<u64>>(s, S{} }
fun is_u64<T>(): bool { exists<S<u64>> }
```

The important property enabling monomorphization is that we can identify such dependencies by looking at the memory accessed by code (and injected specifications). Assume that a function `f<T>` accesses memory `S<T>`. If the same function also access any instantiation of this memory (say `S<u64>`), we need to deal with the case that `S<T>` and `S<u64>` overlap in the effects. Specifically, if we just monmorphize into `f_T` which uses `S_T` and `S_u64`, we miss any conditions dependent on the case that `T = u64`. Consider the following code fragment:

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

This function aborts in the case that `T = u64`, but not necessary if `T != u64`.

The solution to this problem is that we verify not only `f_T` but also `f_u64`. In general, the set of monomorphized instances of a function `f<T>` which need to be verified is determined by finding all instantiations of `T` that some pair of memory accesses in the function can overlap.

TODO(): formalization and proof?

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 common scenario where such code is implicitly created by injection of global invariants. Consider the example in Fig. 9. 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.

3.5 Translation to Boogie and Z3

3.5.1 Vectors and Extensionality

3.5.2 Encoding

3.5.3 Butterflies

4 Application

TODO(wrwg): ...

5 Related Work

TODO(wrwg): ...

6 Conclusion

TODO(wrwg): ...

References

[1] Sam Blackshear, Evan Cheng, David L. Dill, Victor Gao, Ben Maurer, Todd Nowacki, Alistair Pott, Shaz Qadeer, Rain, Dario Russi, Stephane Sezer, Tim Zakian, and Runtian Zho. Move: A Language With Programmable Resources, 2019.

[2] Sam Blackshear, Todd Nowacki, Shaz Qadeer, and John Mitchell. The move borrow checker, 2021.

[3] Dave Clarke, Johan Östlund, Ilya Sergey, and Tobias Wrigstad. Ownership types: A survey. In Dave Clarke, James Noble, and Tobias Wrigstad, editors, *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, volume 7850 of *Lecture Notes in Computer Science*, pages 15–58. Springer, 2013.

[4] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: Securing the foundations of the rust programming language. *Proc. ACM Program. Lang.*, 2(POPL), December 2017.

[5] Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):4051, October 1992.

[6] The Boogie Team. Boogie Intermediate Verification Language.

[7] The CVC Team. CVC5.

[8] The Diem Association. An Introduction to Diem, 2019.

[9] The Diem Association. The Diem Framework, 2020.

[10] The Move Team. The Move Language Definition, 2020.

[11] The Move Team. The Move Specification Language, 2020.

[12] The Z3 Team. Z3 Prover.

[13] Aaron Weiss, Daniel Patterson, Nicholas D. Matsakis, and Amal Ahmed. Oxide: The essence of rust. *CoRR*, abs/1903.00982, 2019.

[14] Jingyi Emma Zhong, Kevin Cheang, Shaz Qadeer, Wolfgang Grieskamp, Sam Blackshear, Junkil Park, Yoni Zohar, Clark Barrett, and David L. Dill. The Move Prover. In Shuvendu K. Lahiri and Chao Wang, editors, *Computer Aided Verification*, pages 137–150. Springer International Publishing, 2020.