Towards Practical Formal Verification
of Smart Contracts
– Technical Report –
DRAFT – Implementation Only View

David Dill   Wolfgang Grieskamp   Junkil Park
Shaz Qadeer   Meng Xu   Emma Zhong
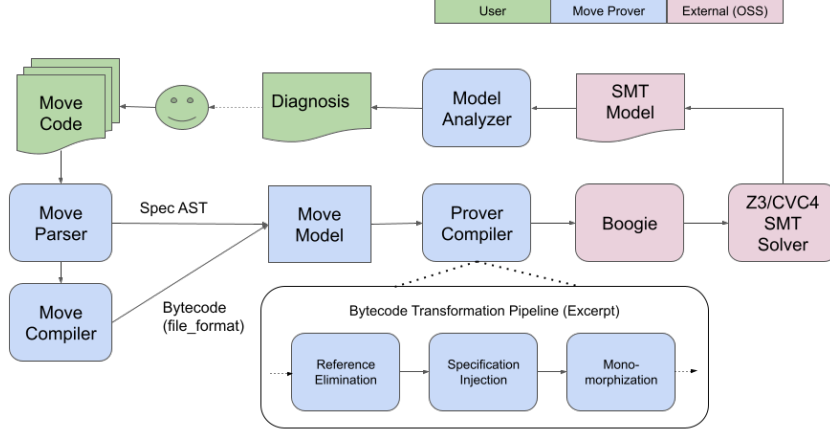
August 25, 2021

Contents

Figure 1: Move Prover Architecture

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. 1. 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.

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 [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 [5] and CVC5 [4]; 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. **??**.) 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* [2] 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 [1]):

- 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. 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 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.

- Similarly, `Mvp::mkglobal(value, TYPE_ID, addr)` creates a new mutation for a global with given type and address. Notice that in the current

4

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 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 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

---

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

Figure 2: 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  }
```

3.3   Function Condition Injection

During 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 section, we cover how *function specification conditions* are injected.

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

6

modules in the dependency relation which are not target of verification have already successfully verified. This is possible since Move has an acyclic import relation.

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. 3.4. 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 Post conditions

The injection of basic function specifications is illustrated in Fig. 3. 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 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 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. 3 does not need it. Snapshots of state will be discussed for global update invariants in Sec. 3.4.

Figure 3: 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  }
```

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 injection is illustrated in Fig. 4.

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 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.

Figure 4: 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  }
```

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.

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

9

Figure 5: Data Invariant Injection

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

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

3.4   Global Invariant Injection

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 step wise.

3.4.1   Basic Translation

Fig. 6 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.

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 functions it is determined from the code. Assuming that the prover has precise knowledge (up to symbolic address representation) of memory usage, it can determine which invariants to inject. Let f be a target function:

- Inject **assume** I at entry to f *if* read*(f) has overlap with read*(I).

- At every point in f where a memory location M is updated inject **assert** I after the update *if* M in read*(I). Also, if I is an update invariant, before

10

Figure 6: Basic Global Invariant Injection

```
1    fun f(a: address) {
2      let r = borrow_global_mut<S>(a);
3      r.value = r.value + 1
4    }
5    invariant [I1] forall a: address: global<S>(a).value > 0;
6    invariant [I2] update
7        forall a: address: global<S>(a).value > old(global<S>(a).value);
8    ⤳
9    fun f(a: address) {
10     spec assume I1;
11     Mvp::snapshot_state(I2_BEFORE);
12     r = <increment mutation>;
13     spec assert I1;
14     spec assert I2[old = I2_BEFORE];
15   }
```

the update inject a memory snapshot save.

Notice that we do not inject any invariants in functions which are not verification target. However, the set of target functions may need to be extended because of invariants, as described later.

3.4.2 Genericity

Figure 7: Genericity

```
1    invariant [I1] global<S<u64>>(0).value > 1;
2    invariant<T> [I2] global<S<T>>(0).value > 0;
3    fun f(a: address) { borrow_global_mut<S<u8>>(0).value = 2 }
4    fun g<R>(a: address) { borrow_global_mut<S<R>>(0).value = 3 }
5    ⤳
6    fun f(a: address) {
7      spec assume I2[T = u8];
8      <<mutate>>
9      spec assert I2[T = u8];
10   }
11   fun g<R>(a: address) {
12     spec assume I1;
13     spec assume I2[T = R];
14     <<mutate>>
15     spec assert I1;
16     spec assert I2[T = R];
17   }
```

In the case of generic invariants and functions, we must use *type unification* to determine which invariants are injected. Consider the example in Fig. 7. Invariant I1 holds for a specific type instantiation S<**u64**>, whereas I2 is generic over all type instantiations for S<T>.

11

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 param-
eterized 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. 3.5.

3.4.3 Modularity

Figure 8: Modular Verification and Invariants

```
 1   module Store {
 2     struct T has key { x: u64 }
 3     public fun read(): u64 { borrow_global<S>(0).x }
 4     public fun write(x: u64) { borrow_global_mut_<S>(0).x = x }
 5   }
 6   module Actor {
 7     use Store;
 8     invariant global<S>(0).x > 0;
 9     public fun update(x: u64) {
10       if (x == 0) then abort 1;
11       Store::set(x);
12     }
13   }
```

In Sec. 3.3.1, the general mechanism of modular verification was described,
deriving the set of verified *target functions* from the set of *target modules*, pro-
vided by the user on the command line. Global invariants add additional func-
tions by possibly requiring re-verification of non-target functions which can in-
fluence the invariant.

Consider the example in Fig. 8. The module Store provides an API for
some storage location which is shared between a set of modules. The module
Actor, one of those modules, establishes an invariant on the content of the
store. When Actor is verified, one must also verify the function Store::write,
because this invariant is verification target. (In this example, verification cannot
succeed, because the function Store::write is not restricting the values for the
parameter x; we see in the next section how to fix this.)

In general, the set of additional functions to verify is computed as follows.
Let I be a target invariant which appears in some target module, and f some
function in the dependency relation. If modify(f) has an overlap with read*(I)

then f needs to be added to the target functions. Notice it is not modify*(f); only direct modifications make a function to a verification target (with exceptions as discussed in the next section).

3.4.4 Suspending Invariants

Figure 9: Suspension of Invariants

```
1   module Store {
2     friend Actor;
3     ...
4     public(friend) fun write(x: u64) { borrow_global_mut_<S>(0).x = x }
5     spec write { pragma suspend_invariants; }
6   }
7   module Actor {
8     ...
9     invariant [suspendable] global<S>(0).x > 0;
10  }
```

The example in Fig. 8 is not quite right from a design viewpoint, since a global store accessible to everybody is constrained by a specific module. Consequently, it cannot be successfully verified. Fig. 9 modifies the example to fix this. First, Move's **friend** mechanism is used to restrict visibility of Store::write to the Actor module. Note one could add other modules to the friends list as needed. Second, the Store::write function is declared to *suspend invariant evaluation to callers*. Only private and friend functions can have such a declaration, ensuring the all call sites are known and the suspended invariants are actually verified in all call contexts. An invariant needs to be explicitly marked via [suspendable] do be eligible for suspension.

When an invariant I is suspended for a function F, the injection scheme changes as follows:

- At the definition side of F, I is neither assumed nor asserted.

- At every call side of F (whether opaque or inlined), the invariant is asserted right after the call. It will also be assumed at the entry point of the caller.

- Instead of F becoming a target function because it modifies the memory read in I (see above paragraph about modular verification), all callers will become target functions.

- If the caller is itself suspended, the process is instead continued with the parent callers.

Once a function is suspended, automatically all functions it calls which modify memory effected by the suspended invariants are suspended as well. This is because when those functions are called, the relevant invariants cannot be

assumed to hold, and therefore it is likely not fruitful to try to verify something related to them.

For update invariants it should be noted that suspension may change their meaning, depending on the form of the predicate. Without suspension, an update invariant is implemented by snapshotting the memory before the update and then asserting a predicate after the update which refers to the previous state and the current one. For suspended update invariants, the snapshot is taken *before* the suspended function is called, and the assertion injected *after* it returns, which might be earlier resp. later states. An example of an update invariant which works well for suspension is e.g. a requirement for a monotonically increasing value, as in **invariant** [suspendable] **old**(value()) <= value().

Methodologically, the suspension mechanism should be used with care, because it may complicate the verification problem by propagating verification errors to more complex application contexts. The Move prover supports a further pragma to suspend invariant verification which draws a clear boundary to function systems with suspension. With **pragma** suspend_invariants_in_body a function can be marked to suspend invariants only in its body but ensure they hold at caller side. This is conceptually syntactic sugar for introducing a helper function:

```
public fun f(P) { S } spec f { pragma suspend_invariants_in_body; }
transform
public fun f(P) { f'(P) }
fun f'(P) { S } spec f' { pragma suspend_invariants; }
```

3.4.5 Invariant Consistency

TODO(wrwg): Describe solution to the below problem via induction

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.

3.5 Monomorphization

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

3.5.1 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

Figure 10: Basic Monomorphization

```
1   fun f<T>(x: T) { g<S<T>>(S(x)) }
2   ⤳
3   struct given_T{}
4   fun f_T(x: given_T) { g_S_T(S_T(x)) }
```

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.

3.5.2   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. 7. 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.

3.6 Translation to Boogie and Z3

3.6.1 Vectors and Extensionality

3.6.2 Encoding

3.6.3 Butterflies

References

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

[2] 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.

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

[4] The CVC Team. CVC5.

[5] The Z3 Team. Z3 Prover.