# Formal Verification of Imperative First-Class Functions in Move

Wolfgang Grieskamp* and Teng Zhang and Vineeth Kashyap and Jake Silverman
Aptos Labs, Palo Alto, USA
{wg,teng,vineeth,jake.silverman}@aptoslabs.com
*Corresponding author

***Abstract*—The Move Prover (MVP) is a formal verifier for smart contracts written in the Move programming language. Recently, Move on Aptos was extended with *higher-order functions*: imperative functions as first-class values that can be passed around, stored in data structs, and kept in persistent storage, enabling *dynamic dispatch*. This paper describes the representation of function values in the Move specification language and their implementation in MVP. We introduce *behavioral predicates* which characterize Move functions (aborts and pre-/postconditions) by single-state or two-state predicates. We also introduce *state labels* for naming intermediate memory states in which expressions are evaluated and which allow to compose behavioral predicates to describe sequences of state transitions. On SMT level, function values are encoded by discriminating over the possible function values reaching a call site: when the concrete function is known, its effect is accounted for directly; when it is unknown (for example, a function parameter, or a closure loaded from storage), its behavioral predicates describe the effect. Our approach goes beyond, for example, Dafny, by supporting imperative first-class functions which can modify state via Rust-style references and global variables, and leads to more efficient SMT encodings than separation logic because of the static separation of memory enabled by Move. We further extend MVP's *specification inference* tool to work with function values: given arbitrary higher-order Move code, weakest-precondition analysis semi-automatically derives behavioral-predicate-based specifications, reducing the annotation burden and providing a validation pipeline for the new specification constructs.**

## I. Introduction

The Move Prover (MVP) [1] is a formal verification tool for Move smart contracts [2], [3], designed for routine use during development with response times comparable to type checkers and linters. MVP is deployed at Aptos, a public layer-1 blockchain whose smart contracts are written in Move, where it verifies core protocol logic — staking, metering, code deployment, and supporting data structures [4], [5]; it was originally developed for the Libra/Diem blockchain at Meta.

Recently, Move was extended with *higher-order functions* [6]: functions are now first-class values that can be passed, returned, stored in struct fields, and persisted in global storage. Combined with Move's resource model, persisted function values give a principled account of *dynamic dispatch*. Typical Aptos use cases include callback-based asset flows, extensible asset managers, dispatchable permissions and metering, and generic data-structure operations parameterized by a caller-provided function (e.g. fold, map, filter, custom comparators).

Higher-order functions are a known challenge for verification: they break the modular, static-dispatch specification style on which MVP was built. When a call is dispatched dynamically, its pre- and post-conditions cannot in general be looked up at the call site; the caller's specification must instead refer *abstractly* to the behavior of its function-value arguments, and the verifier must reconcile this with whichever concrete function reaches the call site. For a production-oriented prover, this reconciliation must be fast, complete in the relevant cases, and free of the trigger instability ('butterfly' effect) that SMT encodings easily introduce [7].

**Contributions.** This paper describes the extension of MVP to support higher-order Move, and the underlying extension of the Move specification language:

- We extend Move specifications with *behavioral predicates* $\mathsf{requires\_of}\langle f\rangle(\bar{x})$, $\mathsf{aborts\_of}\langle f\rangle(\bar{x})$, and $\mathsf{ensures\_of}\langle f\rangle(\bar{x},\bar{y})$, which characterize the pre-/abort-/post-behavior of a function value $f$ on arguments $\bar{x}$ and results $\bar{y}$, making function specifications first-class. They implicitly track read and read/write state dependencies.
- We introduce *state labels*, which name intermediate memory states along a function's execution and connect them via two-state predicates, as in $S1..S2 \mid\sim \phi$, enabling reasoning about sequential evolution of state.
- We describe how MVP implements invocation of function values on top of SMT [8], [9] by discriminating over the function-value variants flowing into the call site: if $f$ is known, the invocation is replaced by the effect of the concrete function; otherwise (e.g., a function-typed parameter or a closure loaded from storage), the behavioral predicates of $f$ describe its effect.
- We extend MVP's *specification inference* tool [10] to function-valued code. The tool computes weakest preconditions over Move bytecode with closures and dynamic dispatch, automatically producing behavioral-predicate-based specifications. This both reduces the annotation burden and yields a validation pipeline confirming the encoding is internally consistent.

While these aspects have similarities to features and solutions in Dafny, F*, or Verus, they are novel in the details. For example, Dafny has state labels but does not support higher-order functions with side effects, nor quantification over state

https://doi.org/10.34727/2026/isbn.978-3-85448-093-8_57

labels. See Sec. V-A for a more thorough discussion of related work.

## II. Using Function Values in Move Specifications

This section recalls the aspects of Move on Aptos [3] that matter for the verification of higher-order programs, introduces an automated market maker (AMM) as a running example, and illustrates the common specification patterns our extension supports. All examples can be also found at [11].

### A. Move on Aptos

Move is a bytecode-verifiable programming language designed from the outset to coexist with formal verification. For the purposes of this paper, the relevant properties are:

- *Strong static typing with generics.* Every expression has a statically known explicit type; generic functions and structs are monomorphized for verification [1].
- *Type-indexed global storage.* A struct type with the `key` ability is called a *resource* and can be stored in global state, which is a map keyed by (resource type, address) pairs: a resource can be moved to, borrowed, and moved from the location `T[addr]`, with linear ownership enforced statically, and `∃<T>(addr)` tests its presence. Specifications quantify over this memory directly, and `modifies` clauses name the footprint a function may touch, providing a frame condition.
- *Deterministic execution.* A Move function's results and effects are fully determined by its arguments and the pre-state of global storage; there is no in-language concurrency or other source of non-determinism.
- *Reference semantics.* References are borrowed from local variables or from global storage; this yields an alias-free memory model that makes verification tractable [1].
- *Strongly-typed bytecode.* Move source is compiled to bytecode whose type- and memory-safety are checked before execution. MVP verifies this bytecode against specifications in the source.
- *Integrated specification language.* Move lets developers write pre-/postconditions, abort conditions, data invariants, and global memory invariants in the same source file as the code they describe, or optionally, into separate files.

**The Move Prover.** MVP is an automated formal verifier built into the Move toolchain; the version discussed here and described in [1] is a complete rewrite of the earlier version in [12]. Developers annotate Move source with specifications — preconditions, postconditions, abort conditions, data invariants, and loop invariants — using a first-order specification language that shares Move's type system. MVP translates the annotated program to Boogie [13], [14], an intermediate verification language, which Z3 [9] then discharges as SMT queries. The tool targets routine use during development: verification is designed to complete within seconds per function through careful SMT encoding and monomorphized, modular function summaries; it is used in CI (continuous integration testing) in the workflows at Aptos. Abort conditions deserve special note: multiple `aborts_if` clauses form a *biconditional* — the function aborts if and only if at least one clause holds — rather than the one-directional "if it aborts, then ..." familiar from partial-correctness Hoare logic.

**Function values in Move.** A new version of Move extends the type system with *function types* $T_1 \times \cdots \times T_n \to T$ (written `|T1,...,Tn|T` in the concrete syntax) and allows values of such types to be constructed from concrete function symbols or from lambda expressions, possibly with captured arguments. In Move's type ability system, function values may be marked `copy` (may be duplicated), `drop` (may be discarded), and `store` (may be placed in struct fields and thereby persisted to global storage). References may not be captured; only plain values may appear in a closure's payload, ensuring that captured data is independent of any borrow scope.

### B. Automated Market Maker Example

The running example throughout this paper is a simplified AMM (Automated Market Maker) module in which the *pricing curve* is a function value stored in the AMM pool. An AMM holds reserves of two assets in a pool and lets traders exchange one for the other, the price determined by a curve over the current reserves rather than by an order book; prominent protocols include Uniswap [15], Curve, and Balancer. The pricing function has to satisfy a number of invariants: (i) the function never aborts, (ii) the output never exceeds the output reserve, (iii) the output is monotone in the input amount, and (iv) the product of the two reserves does not decrease after a swap (constant-product preservation). The spec expresses these invariants using *behavioral predicates* and *state labels*, two constructs introduced by this paper:

```
struct Pool has key {
  reserve_x: u64, reserve_y: u64,
  // (reserve_in, reserve_out, amt) = out
  pricing: |u64, u64, u64|u64 has copy, store
}
spec Pool {
  // Function allowed to read any global state.
  reads_of<self.pricing> *;

  // Pricing must never abort.
  invariant ∀ S in *, ri: u64, ro: u64, a: u64:
    S |~ !aborts_of<self.pricing>(ri, ro, a);

  // Output never exceeds the output reserve.
  invariant ∀ S in *, ri: u64, ro: u64, a: u64:
   S.. |~ result_of<self.pricing>(ri, ro, a) ≤ ro;

  // Monotonicity in the input amount.
  invariant ∀ S in *, ri: u64, ro: u64,
                      a1: u64, a2: u64:
    S.. |~ a1 ≤ a2 ⟹
      result_of<self.pricing>(ri, ro, a1)
           ≤ result_of<self.pricing>(ri, ro, a2);

  // Constant-product preservation.
  invariant ∀ S in *, ri: u64, ro: u64, a: u64:
    S.. |~
      (ri + a)
      * (ro - result_of<self.pricing>(ri, ro, a))
        ≥ ri * ro;
}
```

Each invariant is universally quantified over a state label `S`, ranging over all states reachable during execution in which

the pool exists at some address. A *state label* such as `S` names a program point; the state at `S` comprises both the global memory state and the local state (via references) at that point. For the Pool invariants the local component is not relevant, since `pricing` takes only plain value arguments and no references; the global component covers the resources declared by `reads_of<self.pricing> *`.

State labels appear in two forms. A *single-state* predicate $S \mid\sim p$ evaluates $p$ entirely in state $S$; the no-abort invariant uses this form because whether `pricing` aborts depends only on the pre-state of the call. A *two-state* predicate $S.. \mid\sim q$ takes $S$ as the pre-state of the `pricing` invocation and the resulting post-state as the second end; the output, monotonicity, and constant-product invariants use this form because the return value of `pricing` is only defined after execution from $S$. More generally, $S_1..S_2 \mid\sim q$ names both ends explicitly; in the example, $S.. \mid\sim q$ fixes $S$ as the pre-state, with the post state unspecified.

The behavioral predicates `aborts_of`, `result_of`, and (elsewhere) `ensures_of` and `requires_of`, take a function value and its arguments and evaluate to the corresponding aspect of that call's behavior. Without behavioral predicates, specifying a higher-order function would require inlining the callee's conditions at every call site, leading to a combinatorial explosion in specification size; behavioral predicates make reasoning about function values modular and abstract. Using them in invariants as in the example creates (i) a requirement for any code creating a pool to pass only compliant function values, and (ii) a guarantee for any code calling a pool function that the requirements are met. Both conditions are verified by MVP.

Notice that the monotonicity invariant relates two invocations of the same function value — a *relational* property (hyperproperty). This is no special case: per state pair, `result_of` is a mathematical function of the function value and its arguments, so any first-order formula relating multiple `result_of` terms is a legal specification. For example, `S.. |~result_of<f>(x)> result_of<g>(x)` compares two function values pointwise; relations across different states are expressed via distinct state labels.

**Swapping.** The major functionality of the pool is to facilitate `swap` (exchange) of one asset with another.

```
fun swap(pool: &mut Pool, amt: u64): u64 {
  let out = (pool.pricing)
            (pool.reserve_x, pool.reserve_y, amt);
  pool.reserve_x += amt; pool.reserve_y -= out;
  out
}
spec swap {
  // Can only abort on reserve overflow.
  aborts_if pool.reserve_x + amt > MAX_U64;
  // Outcome is determined by pricing function.
  ensures result = old(result_of<pool.pricing>(
            pool.reserve_x, pool.reserve_y, amt));
  // Preserves product in pool.
  ensures pool.reserve_x * pool.reserve_y
       ≥ old(pool.reserve_x * pool.reserve_y);
}
```

In a postcondition, `result` is the predefined name of the function's return value, and `old(e)` evaluates `e` in the function's pre-state. The specification of `swap` can be verified by MVP because of the invariants guaranteed for the (dynamic) `pricing` function. In this case, we only illustrate the aborts condition and product preservation. For aborts, `pricing` itself is non-aborting, and (more subtly), `pool.reserve_y - out` cannot underflow because of the 'output never exceeds the output reserve' invariant, so only the overflow condition remains. (Notice MVP would flag any aborts conditions not covered). For product preservation, the constraint can be derived as well.

**Pricing variants.** At the time a pool value is constructed, the invariants for the pricing function need to be verified. Obviously this is more difficult than the application side, since now properties of functions need to be proven. In reality, those proofs will often need custom solutions, but for the given example, MVP can handle some cases gracefully. First, we define a function which computes the so-called *constant product*, which is a pricing function describing a direct swap without fees; however, rounding errors can lead to a loss in returned assets:

```
fun product(ri: u64, ro: u64, a: u64): u64 {
  let num = (ro as u128) * a;
  let den = (ri as u128) + a;
  if (den = 0) 0 else (num / den) as u64
}
```

When constructing a pool using `Pool{.., pricing: product}`, MVP is able to discharge all verification conditions.

The next example uses a pricing function which charges a configurable fee:

```
struct Fee has key {bps: u64}
fun with_fee(owner: address,
             ri: u64, ro: u64, a: u64): u64 {
  let b = (a as u128)
     * (10000 - Fee[owner].bps as u128) / 10000;
  product(ri, ro, b as u64)
}
spec with_fee {
  aborts_if !∃<Fee>(owner);
  aborts_if Fee[owner].bps > 10000;
  ..
}
```

In order to construct a pricing function we need to curry the address for the fee resource and capture it in a closure, which can be done in Move as below:

```
  let owner: address;
  Pool{
    ..,
    pricing: |ri, ro, a|
      with_fee(owner, ri, ro, a)
  }
```

MVP detects that `with_fee` violates the requirement to not abort. Guarding the definitions with checks and using a default fee fixes this and makes verification succeed.

```
fun with_fee_compliant(owner: address,
             ri: u64, ro: u64, a: u64): u64 {
  let bps =
    if (!∃<Fee>(owner)
        || Fee[owner].bps > 10000)
```

```
        500
      else
        Fee[owner].bps;
    let b =
        (a as u128) * (10000 - bps as u128) / 10000;
    product(ri, ro, b as u64)
  }
  spec with_fee_compliant {
    aborts_if false;
    ..
  }
```

Notice that currently, a user needs to provide complete aborts conditions of even simple functions as `with_fee`. While MVP in general allows to keep aborts conditions unspecified this does not work right now for aborts conditions in the case of function values. We are looking at connecting specification inference in MVP to the use case of compliance for function values; see also Sec. IV.

The AMM example shows that non-trivial invariants can be specified for function values, and that the prover is capable of verifying them on the construction and caller sides, even though the invariants presented constitute non-linear arithmetic problems. While determining that the aborts contract is violated by `with_fee` appears easy (after all, one can directly see in the spec that the function aborts), there are more subtle things happening in the background – for example, that `product` does not have any underflow on subtraction because of the `pricing` invariant that the return value does not exceed the reserve.

However, we have no illusions about the scalability of compliance-side verification. In general, passing a function $f$ as an actual parameter $p$ requires proving $\forall x$ : requires_of$\langle p\rangle(x) \Rightarrow$ requires_of$\langle f\rangle(x)$, $\forall x$ : aborts_of$\langle f\rangle(x) \Leftrightarrow$ aborts_of$\langle p\rangle(x)$, and $\forall x, y$ : ensures_of$\langle f\rangle(x, y) \Rightarrow$ ensures_of$\langle p\rangle(x, y)$; Sec. III-D describes how these obligations materialize in the encoding. For the examples in this paper, `product` and the application side (`swap`, the callers of `find`) verify from the shown specifications alone; the nonlinear fee variants need in addition a one- or two-line proof hint (a case split, and a stepping-stone assertion bounding the fee-adjusted input). For harder universal properties of functions, we do not expect compliance proofs to be automated easily; rather, they will require more substantial proof hints and/or trusted assumptions. Nevertheless, on the application side of function values, such as inside the `swap` function, verification is not more complex than for first-order functions.

### C. Filter-style Loops

Behavioral predicates are not limited to struct invariants; they appear naturally in loop invariants and function specs for higher-order collection operations. As a minimal illustration, consider a generic `find` which returns the first index of `v` whose element satisfies a predicate `pred`, or `len(v)` when no such element exists. A spec-level helper `no_match_before` captures the recurring "no earlier element has matched" pattern:

```
fun find<T>(v: &vector<T>,
            pred: |&T|bool has copy + drop): u64 {
  let i = 0;
  let n = v.length();
  while (i < n) {
    if (pred(v[i])) return i;
    i += 1;
  } spec {
    invariant i ≤ n;
    invariant no_match_before(v, pred, i);
  };
  n
}
spec find {
  // opaque: caller side uses spec only
  pragma opaque;
  ensures result ≤ len(v);
  ensures no_match_before(v, pred, result);
  ensures result < len(v)
            ⟹ result_of<pred>(v[result])
}
spec fun no_match_before<T>(v: vector<T>,
              pred: |&T|bool, end: u64): bool {
  ∀ j in 0..end: !result_of<pred>(v[j])
}
```

`no_match_before` is an ordinary spec function, but its body calls the behavioral predicate `result_of<pred>` on each element — a predicate-parametric abstraction is thus first-class in Move and can be reused across loop invariants and post-conditions. The function's post-condition characterizes `result` entirely through the same evaluator: either `result` is a valid index whose call is the first to match, or `result` $=$ `len(v)` and no element matches at all. No concrete predicate is ever named in either the body or the spec; a caller instantiates `pred` with any function value, and the specification is discharged through its behavioral contract.

To illustrate the proof obligation at the caller, consider specializing `find` with an inline lambda whose behavioral contract is given by an inline `spec` clause. As was discussed earlier for `with_fee`, functions provided as parameters need to have explicit specifications, therefore the lambda has a trivial spec block:

```
fun find_zero_lambda(v: &vector<u64>): u64 {
  find(v, |x| x = 0
           spec { ensures result = (x = 0) })
}
spec find_zero_lambda {
  ensures result ≤ len(v);
  ensures result < len(v) ⟹
    v[result] = 0 &&
    (∀ j in 0..result: v[j] ≠ 0);
  ensures result = len(v) ⟹
    (∀ j in 0..len(v): v[j] ≠ 0);
}
```

Since `find` is opaque, MVP reasons from `find`'s spec alone. (Without opaque, MVP uses spec and body combined at caller side.) At the match index, `find` guarantees result_of$\langle pred\rangle(v[result])$, which — by the lambda's spec — reduces to $v[result] = 0$. For earlier indices, `no_match_before` constrains result_of$\langle pred\rangle(v[j])$ to be false for every index $j$ before *result*, which reduces to $v[j] \neq 0$. MVP discharges all three `ensures` clauses of `find_zero_lambda` without ever inspecting the lambda body.

Effectively, with the higher-order function `find`, we have verified a summary of this specific loop pattern which can now be reused many times on the caller side. Users calling functions like `find`, `map`, `reduce`, etc. will hardly need to write loop invariants themselves.

### D. More About State Labels

The Pool invariants in Sec. II-B used *universally* quantified state labels: every state in which the pool exists must satisfy the pricing invariants. State labels have a second, equally important use: *existentially* quantifying over an intermediate state to express that a computation proceeds through a specific intermediate point. Consider a higher-order function that sequences two stateful operations on a shared value:

```
fun followed_by(f: |&mut A|, g: |&mut A|, x: &mut
    A) {
  f(x); g(x)
}
spec followed_by {
  reads_of<f> *; reads_of<g> *;
  ensures ∃ S in *:
    (..S |~ ensures_of<f>(x)) &&
    (S.. |~ ensures_of<g>(x))
}
```

The existentially quantified label `S` witnesses the intermediate state between the two calls. Since no `modifies_of` is declared for `f` or `g`, they are assumed to have no global memory modification; the local component of the state at `S` is therefore the updated value of `x`, and the global component is unchanged. The conjunct $..S \mid\sim \mathsf{ensures\_of}\langle f\rangle(x)$ asserts that the postcondition of `f` holds from the pre-state of `followed_by` up to `S`; the conjunct $S.. \mid\sim \mathsf{ensures\_of}\langle g\rangle(x)$ asserts that the postcondition of `g` holds from `S` to the final post-state. Crucially, the post-state of `f` automatically becomes the pre-state of `g`, without any explicit threading of intermediate values through the specification.

State labels are not restricted to higher-order functions. When `f` and `g` are concrete named functions, the same pattern allows a wrapper to be specified by composing the behavioral predicates of its callees, avoiding duplication of their postconditions.

## III. Encoding

We describe the implementation of function values as a translation into a conceptual (pseudo-code style) imperative intermediate verification language (IVL) inspired by Boogie [16], [14]. The encoding is given at the IVL level because this is where the interesting design decisions become visible — the subsequent step to pure SMT is straightforward and uses standard techniques. The fragment of the IVL used in this section is familiar: procedures with **havoc**, **assume**, **assert**, pure functions, *algebraic datatypes* (tagged unions with selectors, like Rust enums), universal *axioms* with *triggers*, and polymorphic map types. State is represented by Boogie-style maps; in particular, each resource type $\tau$ yields a map $M_\tau \in \mathsf{address} \to \tau$, with address being the domain of addresses in Move (256-bit integers). The representation is similar to that in the earlier MVP paper [1].

In the following we focus on the aspects specific to function values: the representation of function values (Sec. III-A), the memory access and modification sets used to define the invocation schema (Sec. III-B), the encoding of invocation of function values (Sec. III-C), the encoding of behavioral predicates via axioms (Sec. III-D), and finally the encoding of intermediate state labels (Sec. III-E). Everything in this section operates *after* monomorphization, so type parameters have been specialized and the set of closure, parameter, and field variants reaching a given function type is known statically [1].

### A. Representing Function Values

For each function type $\tau = (T_1, \ldots, T_n) \to T$ that occurs in the monomorphized program, we create a datatype in the IVL which represents all the potential *sources* of function values in a given program. To this end, MVP's mono-analysis records three sets: the set $\mathcal{C}_\tau$ of concrete functions $f$ constructing closures of type $\tau$, the set $\mathcal{P}_\tau$ of function-typed parameters of verification targets, and the set $\mathcal{F}_\tau$ of storable function-typed struct fields whose type is $\tau$. The union of these three sets is materialized as an algebraic datatype in the IVL — one constructor per variant:

$$\begin{array}{l}\textbf{datatype}\ \tau\ \{\\ \quad C_f(\overline{c:T}), \ldots, P_{f,x}(), \ldots, F_{\sigma.x}(n:\mathbb{N}), \ldots\\ \}\end{array}$$

Here $\overline{c:T}$ are the types of the values captured by closure $f$; $P_{f,x}$ is the parameter variant for function-value parameter $x$ of function $f$; and $F_{\sigma.x}$ is the field variant for struct field $x$ of struct type $\sigma$ (we use $\sigma$ here to avoid confusion with the state-label variable $S$ introduced in Sec. III-E). Parameter variants are nullary because, in a well-typed verification condition (VC), the value of an entrypoint function-typed parameter is a skolem constant pinned to its variant at entry; no runtime payload is required. Field variants carry an integer discriminator $n$ so that two locations in memory holding distinct field values are not forced to be equal — a datatype with a single nullary constructor would otherwise collapse all field values to one.

Notice that this representation captures *all* possible functions of type $\tau$ relevant for a given VC. So we do not actually need to deal with an arbitrary number of function values when it comes to higher-order function verification, but only those which can be derived from the program in a given context. The representation is further narrowed if the function value of a given type is not stored in a structure which ends in global memory. Storing a function value in a struct field introduces dynamic dispatch: two different field locations may hold different closures, and the discriminator $n$ in $F_{\sigma.x}(n)$ ensures the solver treats them as distinct. Nevertheless, we are able to pinpoint a specification to a particular $F_{\sigma.x}(n)$, as illustrated by the AMM example (Sec. II-B).

### B. Computing Memory Accesses

Invoking a function value requires reasoning about which memory maps a given function type $\tau$ may read ($\mathcal{M}_\tau$) and

which locations it may modify ($\Delta_\tau(\overline{p})$); we define both here before presenting the invocation schema. Intuitively, the access set is the *read footprint* of the function values of type $\tau$ — it delimits which parts of memory their behavior may depend on — while the modification set is their *write footprint*, providing the frame condition: everything outside of it is unchanged by an invocation. To recall, in the Move language, persistent global storage, or memory, can be accessed via type indexing. For instance, if `R` is a resource type, we write `&mut R[addr]` in the language to obtain a mutable reference to the memory which can then be subsequently mutated.

MVP's representation of memory uses type-indexed maps $M_\rho \in \mathsf{address} \to \rho$, with $\rho$ a resource type (a `struct` in Move with the `key` ability). In the imperative procedures of the IVL, those maps are implicitly available as global variables. But in pure expressions like behavioral predicates, they need to be made explicit as parameters; this is why $\mathcal{M}_\tau$ appears as an explicit argument to behavioral predicates throughout this section.

A few definitions for memory maps are needed. Note that these are definitions on the meta level of the description here and not part of the IVL; they are used to describe pseudo code in terms of the IVL.

First, with $\mathcal{M}_R = \{M \in \mathsf{address} \to \rho \mid \rho \in R\}$ we denote a set of memory maps for the resource types in $R$. The $R$ can be omitted if it is clear from the context. For convenience, we write $\mathcal{T}(\mathcal{M})$ for the set of all types $\rho$ used in the memory maps of $\mathcal{M}$. With $\mathcal{M}_R \downarrow S$, where $S \subseteq R$, we denote the projection $\{M \in \mathcal{M}_R \mid \rho(M) \in S\}$.

In addition to memory, we also need to reason about modifications. A modification is a set of pairs of a type and an expression of the IVL of type address, $\Delta \in \mathbb{P}(\mathcal{T} \times \mathcal{E})$. We write $\Delta(\overline{p})$ for a set of modifications built on expressions referencing parameters in $\overline{p}$. For instance, the declaration `modifies_of<f>(a: address)R[a]` (introduced below) contributes the pair $(R, a)$; at an invocation of `f`, $a$ is bound to the actual argument, so $\Delta(\overline{p})$ gives the modified locations as a function of the arguments $\overline{p}$.

**General Access.** Memory access $\mathcal{M}_\tau$ is computed as the union of all the accesses in the variants of $\tau$, $\mathcal{M}_{C_f} \cup \mathcal{M}_{P_{f,x}} \cup \mathcal{M}_{F_{\sigma.x}}$. For a closure, $\mathcal{M}_{C_f}$ can be statically derived from the code. For parameters and fields, the information is declared in the spec block:

```
spec foo(f: |T|S, g: |T|S) {
  reads_of<f> R; // f can read resource R
  reads_of<g> *; // g can read any resource
                 // in scope of the VC
}
```

For the wildcard operator `*`, all memory which is read or written as part of the currently generated VC is included, plus an abstract memory map for any 'unknown' memory the function may depend on. The unknown memory is needed for a sound encoding. For instance, if we verify function `foo` above in a context where no memory is involved anywhere, we must not assume that `g(x)` delivers the same value in different states, as it may depend on this unknown additional memory. However, we can still know that the specification of `g` holds in every possible state.

**Write Accesses.** Similarly to general access, the modifies set $\Delta_\tau$ is computed as the union of all the modifications in the variants of $\tau$, $\Delta_{C_f} \cup \Delta_{P_{f,x}} \cup \Delta_{F_{\sigma.x}}$.

For closures, $\Delta_{C_f}$ is derived from the code and, if present, the `modifies R[a]` declaration in the language. These existed before higher-order functions, for specifying what functions modify.

For parameters and fields, the information is declared in the spec block, where for the `modifies_of<f>` clause, the parameters of the function value itself are passed:

```
spec foo(f: |address|) {
  // f can modify resource R[a]
  modifies_of<f>(a: address) R[a];
}
```

If no `modifies_of` is declared, it is assumed that the function does not modify global memory.

Notice that a wildcard (`modifies_of<f> *`) is not supported, since the invocation of `f` would render the memory arbitrary (no effective frame condition). Consider a generic utility `for_each_account(f: |address|)` applying a caller-supplied, state-mutating callback: one would like to specify it once, for callbacks with open-ended write footprints, but today the footprint must be declared per resource, as in `modifies_of<f>(a)R[a]`. Lifting this restriction is left for future work.

### C. Invoking Function Values

When a function value is invoked, we are looking at either a concrete closure value $C_g$ or a parameter or field selection as described above. In the former case, one can simply delegate to calling the actual underlying function $f$, by composing the captured and provided arguments into a full argument list.

In the other cases of parameters and field selections, we use the behavioral predicates as defined for the function value, using the standard schema for encoding a call to a Move function via the function's specification. This reduces the problem of invocation to how those behavioral predicates are encoded. In the procedure below, $\mathcal{M}_\tau$ denotes the projection of the current global memory state to the resource types in $\tau$'s access set (as computed in Sec. III-B).

$$
\begin{array}{l}
\textbf{proc}\ \mathit{invoke}_\tau(x : \tau, \overline{p : T})\ \mathsf{returns}\ (\overline{r : T}) \\
\quad \textbf{if}\ x\ \mathsf{is}\ \ C_f(\overline{c}) \\
\qquad \overline{r} := \mathsf{call} f(\overline{c}, \overline{p}) \\
\quad \textbf{else} \\
\qquad \textbf{assert}\ \mathit{requires}_\tau(x, \mathcal{M}_\tau, \overline{p}) \\
\qquad \textbf{if}\ \ \mathit{aborts}_\tau(x, \mathcal{M}_\tau, \overline{p}) \\
\qquad\quad \mathit{abort\_flag} := \textbf{true} \\
\qquad \textbf{else} \\
\qquad\quad \textbf{let}\ \mathcal{M}'_\tau := \mathcal{M}_\tau \\
\qquad\quad \textbf{havoc}\ \Delta_\tau(\overline{p}),\ \overline{r} \\
\qquad\quad \textbf{assume}\ \mathit{ensures}_\tau(x, \mathcal{M}'_\tau, \mathcal{M}_\tau, \overline{p}, \overline{r})
\end{array}
$$

Here $\textbf{let}\ \mathcal{M}'_\tau := \mathcal{M}_\tau$ freezes the pre-invocation memory into $\mathcal{M}'_\tau$, while $\mathcal{M}_\tau$ becomes the post-invocation state after

**havoc**. We therefore pass $\mathcal{M}'_\tau$ as the pre-state and $\mathcal{M}_\tau$ as the post-state to $ensures_\tau$ (note that $\mathcal{M}'$ consistently denotes the pre-state throughout this section). $\mathbf{havoc}\,\Delta_\tau(\overline{p})$ is shorthand for havocing, for each $(\rho, a) \in \Delta_\tau(\overline{p})$, the entry $M_\rho[a]$ of the type-indexed memory map for $\rho$. *abort_flag* is a prover-internal boolean global that signals an abort to the enclosing verification condition, consistent with how MVP models Move's abort semantics throughout the IVL. The **if** branch reduces to a standard function call $\mathsf{call} f(\overline{c}, \overline{p})$, which MVP encodes using the same **assert/havoc/assume** schema as the **else** branch, specialized to $f$'s declared spec. When $f$ is opaque, the two branches are in fact semantically equivalent: an opaque call is already compiled to this schema driven by $f$'s spec, matching what the **else** branch does for the abstract variant. The **else** branch therefore introduces no new encoding machinery; it applies the existing opaque-call schema, driven by the behavioral predicates of the unknown variant $x$ rather than a concrete function's spec.

### D. Behavioral Predicates

Behavioral predicates are derived by switching over the variants of the function value representation $\tau$, similarly to the $invoke_\tau$ procedure above, delegating extraction of the underlying predicates to per-variant functions. Intuitively, each behavioral predicate is total over the variants of $\tau$: applied to a closure, it unfolds to the underlying function's specification; applied to an abstract parameter or field variant, it remains an opaque symbol of which only the user-declared conditions are known.

Below, the definition for ensures is given. The other predicates are defined similarly. Specific to ensures is that it needs to take care of the frame condition: the caller havoced the full modification set, but only a subset of that is constrained by the given variant. The handling needs to account for the fact that modification addresses are symbolic expressions that may alias.

In what follows, we write $pred_\tau[\mathcal{M}', \mathcal{M}](\ldots)$ with square brackets for the memory pair: $\mathcal{M}'$ is the pre-state and $\mathcal{M}$ the post-state, following the convention established in Sec. III-C. These memory arguments must be passed explicitly to pure functions, whereas they are implicit globals in IVL procedures. We write $\Delta_x$ for the modification set of the specific variant of $x$ — that is, $\Delta_{C_f}$ when $x = C_f(\overline{c})$, $\Delta_{P_{g,y}}$ when $x = P_{g,y}()$, and $\Delta_{F_{\sigma.x}}$ when $x = F_{\sigma.x}(n)$. We write $\mathcal{M}'(\rho)$ for the map $M_\rho$ in the pre-state, and $\mathcal{M}(\rho)$ for the map $M_\rho$ in the post-state.

$$
\begin{aligned}
&\mathbf{fun}\ ensures_\tau[\mathcal{M}', \mathcal{M}](x : \tau, \overline{p : T}, \overline{r : U}) : \mathsf{bool} \\
&\quad \textstyle\bigwedge_{(\rho,a)\in\Delta_\tau} ( \\
&\qquad \mathbf{if}\ \rho \notin \mathcal{T}(\Delta_x) \\
&\qquad\quad \mathcal{M}(\rho) = \mathcal{M}'(\rho) \\
&\qquad \mathbf{else} \\
&\qquad\quad (\textstyle\bigwedge_{(\rho,a')\in\Delta_x} a' \neq a) \\
&\qquad\qquad \Rightarrow \mathcal{M}(\rho)[a] = \mathcal{M}'(\rho)[a] \\
&\quad ) \wedge ensures_x[\mathcal{M}'_x, \mathcal{M}_x](p, r)
\end{aligned}
$$

The relational representation of a function via ensures does not yet capture that Move functions are *deterministic* (Sec. II-A): for a fixed function value, arguments, and pre-state, there is exactly one outcome. Functional behavior is established by introducing $result_\tau$ as an uninterpreted function, connected to ensures via an axiom (from here on, quantifiers are annotated with their *trigger* in braces, further discussed in Sec. III-F):

$$
\begin{aligned}
&\mathbf{fun}\ result_\tau[\mathcal{M}', \mathcal{M}](x : \tau, \overline{p : T}) \to \overline{U} \\
&\mathbf{axiom}\ \forall x, \overline{p} \bullet \{result_\tau[\mathcal{M}', \mathcal{M}](x, \overline{p})\} \\
&\quad \mathbf{let}\ \overline{r} := result_\tau[\mathcal{M}', \mathcal{M}](x, \overline{p}) \bullet \\
&\qquad ensures_\tau[\mathcal{M}', \mathcal{M}](x, \overline{p}, \overline{r})
\end{aligned}
$$

The result tuple $\overline{U}$ comprises the declared results of $\tau$ and, per mutable-reference parameter, the post-state of the referenced value; thus both the explicit results and the implicitly modified local state are pinned as a function of the function value, its arguments, and memory. For a resource which the variants of $\tau$ only read, a single memory map is passed, so the outcome is determined by the pre-state alone, as determinism demands; the post-state of *modified* global memory remains relationally constrained by the invocation schema (havoc within the frame $\Delta_\tau$, then assuming ensures).

The axiom is deliberately one-directional, stating that the tuple chosen by $result_\tau$ satisfies $ensures_\tau$. The seemingly natural biconditional $ensures_\tau(x, \overline{p}, \overline{r}) \Leftrightarrow \overline{r} = result_\tau(x, \overline{p})$ would force *all* solutions of ensures to coincide — inconsistent as soon as the declared ensures of some variant under-constrains its outcome, such as a mere `ensures result > 0`.

The actual conditions extracted depend on the variant. For closures, they are directly extracted from the spec block. For parameters and fields, we introduce uninterpreted functions; for ensures specifically, we introduce an uninterpreted per-variant result function and *define* the variant's ensures as its graph, so functional behavior holds by construction (the field variant is analogous to the parameter variant shown, with the discriminator $n$ as additional argument):

$$
\begin{aligned}
&\mathbf{fun}\ ensures_{C_f(\overline{c})}[\mathcal{M}', \mathcal{M}](\overline{p : T}, \overline{r : U}) : \mathsf{bool} \\
&\quad \langle \textit{extract from spec block of } f(\overline{c}, \overline{p}) \rangle \\
&\mathbf{fun}\ result_{P_{f,x}}[\mathcal{M}', \mathcal{M}](\overline{p : T}) \to \overline{U} \\
&\mathbf{fun}\ ensures_{P_{f,x}}[\mathcal{M}', \mathcal{M}](\overline{p : T}, \overline{r : U}) : \mathsf{bool} \\
&\quad \overline{r} = result_{P_{f,x}}[\mathcal{M}', \mathcal{M}](\overline{p})
\end{aligned}
$$

The meaning of the uninterpreted functions is determined by how they are injected into a VC via the regular mechanism by which pre-/postconditions and invariants are handled. Consider a precondition like `requires !aborts_of<param>(x)`, where `param` is a function-value parameter of a function `foo`. This will be injected in the VC as $\mathbf{assume}\,\neg\ aborts_{P_{foo,param}}(x)$. Similarly, for a data invariant of a field, an assumption will be generated when the field is selected.

**Compliance checking.** The same mechanism yields the compliance obligations of Sec. II-B when a concrete function value flows into an abstract position. Since behavioral predicates are first-order, compliance verification mimics the conventional handling of pre-/postconditions: the condition `requires ! aborts_of<param>(x)` above, assumed on the definition side

of `foo`, is *asserted* at call sites of `foo`, where `param` is bound to an actual function value — say a closure $C_g(\bar{c})$ — so the assertion reduces, via the per-variant extraction, to a condition over $g$'s specification. Data invariants over function-valued fields, as in the Pool example, create the same obligations at pack time; constraints via $result_\tau$, including relational ones, are discharged through the result axiom. The resulting VCs quantify over the function value's arguments (and possibly memories) and may involve non-linear arithmetic (Sec. II-B); they are inherently harder than application-side VCs and may need proof hints.

### E. Intermediate State Labels

Recall from Sec. II that a state label `S` denotes a program point. A specification mentioning `S` talks about a state *in the middle* of a function's execution, invisible in a pre-/post-state contract; the encoding materializes it as an existentially quantified snapshot, connected to both ends by the conditions mentioning `S`. Formally, a *state* at label $S$ is a pair $(\mathcal{M}^S, \bar{v}^S)$ consisting of a memory snapshot and a tuple of local values. The memory component $\mathcal{M}^S$ has the same type as the ambient memory: a tuple of type-indexed maps $M_\rho \in \mathsf{address} \to \rho$ for each resource type $\rho$ in scope at that program point, quantified existentially in the IVL. The local component $\bar{v}^S$ captures values of mutable-reference parameters at the intermediate point: for a closure over a mutable reference of type $T$ with no global memory effects, $\bar{v}^S$ is a single value $x^S : T$, and $\mathcal{M}^S$ is vacuous (the function touches no global resources). When a closure both accesses global memory and takes mutable references, both components are non-trivial. Here we focus on the global-memory case (vacuous $\bar{v}^S$); Sec. III-F notes how the mutable-reference case is handled.

Consider the following specification using intermediate state labels:

```
struct R(u64) has key;
spec foo(f: |address|, g: |address|, a: address) {
  ensures ..S |~ ensures_of<f>(a)
  ensures S |~ R[a].0 > 0;
  aborts_if S |~ aborts_of<g>(a);
  ensures S.. |~ ensures_of<g>(a)
}
```

As outlined in the last section, we want to be able to extract behavioral predicate functions from this specification. For the ensures we can come up with the following, which simply replicates part of the spec:

$$\begin{aligned}&\textbf{fun}\ \mathit{ensures}_{C_{foo}}[\mathcal{M}', \mathcal{M}](a)\\ &\quad \exists\, \mathcal{M}^S \bullet \mathit{ensures}_\tau(f, \mathcal{M}', \mathcal{M}^S, a) \wedge R[\mathcal{M}^S][a].0 > 0 \wedge\\ &\qquad \mathit{ensures}_{\tau'}(g, \mathcal{M}^S, \mathcal{M}, a)\end{aligned}$$

For the aborts which happens at the intermediate state `S`, the state `S` needs to be first established before the aborts condition can be checked for g. To achieve this, all ensures conditions which reach `S` are combined with the aborts condition, as in:

$$\begin{aligned}&\textbf{fun}\ \mathit{aborts}_{C_{foo}}[\mathcal{M}', \mathcal{M}](a)\\ &\quad \exists\, \mathcal{M}^S \bullet \mathit{ensures}_\tau(f, \mathcal{M}', \mathcal{M}^S, a) \wedge R[\mathcal{M}^S][a].0 > 0 \wedge\\ &\qquad \mathit{aborts}_{\tau'}(g, \mathcal{M}^S, a)\end{aligned}$$

This is semantically sound because we can assume inductively that the function *foo* has successfully verified. For a successfully verified function, where $\overline{A}$ are the `aborts_if` conditions and $\overline{P}$ the `ensures` conditions, it holds:

$$\bigvee \overline{A} \vee \bigwedge \overline{P}$$

That is, either the function aborts under one of the given conditions, or all of its ensures conditions hold.

In general, since the relation between state labels spawned by predicates of the form $S_1..S_2 \mid\sim p$ must be acyclic, one can extract from $\overline{P}$ those conditions which lead into $\mathcal{M}^S$, and from $\overline{A}$ those which start from state $\mathcal{M}^S$. Given this, the aborts condition can be synthesized using the following scheme:

$$\bigwedge \overline{P}_{..\mathcal{M}^s} \wedge (\bigvee \overline{A}_{\mathcal{M}^s} \vee \bigwedge \overline{P}_{\mathcal{M}^s..\mathcal{M}^T} \wedge (\bigvee \overline{A}_{\mathcal{M}^T} \vee \bigwedge \ldots))$$

Here, $\overline{P}_{..\mathcal{M}^s}$ establish state $\mathcal{M}^S$ in which we check for the aborts condition $\overline{A}_{\mathcal{M}^s}$, or recurse over the remaining $\overline{P}$ and $\overline{A}$.

### F. Implementation Notes

A straightforward implementation following the conceptual description above leads to SMT timeouts on non-trivial examples. The root cause is that the presentation writes behavioral predicates as IVL functions with explicit bodies ($\textbf{fun} \ldots = \mathit{body}$), and Boogie unfolds function bodies eagerly: any quantifier inside a body — common in arithmetic identities or prelude vector axioms — spawns trigger instantiations at every call site, causing combinatorial blowup in the SMT search.

We overcame this by adopting a pattern that decouples the *definition* of behavioral predicates from their *use*: declare each predicate as an *uninterpreted* function and connect it to the per-function specification via axioms with explicit, carefully chosen triggers. This pattern is not specific to higher-order functions; it is a general Boogie encoding discipline for modular specifications that keeps SMT instantiation under control. The remaining implementation choices are natural refinements of applying this pattern consistently:

– *Behavioral predicates are uninterpreted functions connected by axioms, not Boogie functions with explicit bodies.* The implementation declares $\mathit{ensures}_\tau$ as an uninterpreted Boogie function and connects it to the per-function spec via an axiom of the form $(\forall \ldots :: \mathit{eval\_call} \Leftrightarrow \mathit{rhs})$ with an explicit trigger pattern. Axioms with explicit triggers fire only when the trigger pattern matches a ground term in the proof context, keeping instantiation under control.

– *Per-variant trigger specialization.* The three variant kinds receive structurally different axioms. Shown for $\mathit{ensures}_\tau$ with the memory parameters elided for brevity, the axiom schemas are:

$$\textbf{axiom}\ \forall \bar{c}, \bar{p}, \bar{r} \bullet \{ensures_\tau(C_f(\bar{c}), \bar{p}, \bar{r})\}$$
$$ensures_\tau(C_f(\bar{c}), \bar{p}, \bar{r}) \Leftrightarrow ensures_{C_f(\bar{c})}(\bar{p}, \bar{r})$$
$$\textbf{axiom}\ \forall \bar{p}, \bar{r} \bullet \{ensures_\tau(P_{f,x}(), \bar{p}, \bar{r})\}$$
$$ensures_\tau(P_{f,x}(), \bar{p}, \bar{r}) \Leftrightarrow ensures_{P_{f,x}}(\bar{p}, \bar{r})$$
$$\textbf{axiom}\ \forall f, \bar{p}, \bar{r} \bullet \{ensures_\tau(f, \bar{p}, \bar{r})\}$$
$$f\ \mathsf{is}\ F_{\sigma.x} \Rightarrow (ensures_\tau(f, \bar{p}, \bar{r}) \Leftrightarrow ensures_{F_{\sigma.x}}(f.n, \bar{p}, \bar{r}))$$

For closure variants, the trigger is the evaluator application *containing* the constructor term $C_f(\bar{c})$, with the captures bound by the quantifier — a bare constructor pattern would not cover the remaining bound variables; for parameter variants, the nullary ground constructor $P_{f,x}()$ plays the same role. For struct-field variants no such constructor term exists: at call sites a field-stored function value is an opaque datatype term loaded from memory. The trigger is therefore the evaluator application itself, with the variant pinned by the $f$ is $F_{\sigma.x}$ guard and the discriminator passed via the selector $f.n$. In each case the axiom is instantiated only where the variant — or, for fields, the evaluator application — is syntactically present in the proof context.

– *Constraining state-label conjuncts are dropped from the existential.* The conceptual behavioral predicate for an `aborts_if` at an intermediate state $\mathcal{M}^S$ includes *all* ensures-with-$S$ conjuncts that reach $\mathcal{M}^S$ (including constraining ones such as $R[\mathcal{M}^S][a].0 > 0$). The implementation restricts the existential body to the *defining* conjuncts — those introduced by label-defining operations like $\mathsf{ensures\_of}\langle f \rangle(\bar{x})$, $\mathsf{result\_of}\langle f \rangle(\bar{x})$, or the publish/remove/update builtins. By the validity invariant $\bigvee \bar{A} \vee \bigwedge \bar{P}$, the constraining conjuncts hold at the witness chosen during verification of `foo`, so dropping them is sound and keeps the existential body small.

– *State-label existential is flat, not recursive.* The recursive scheme above is logically equivalent to a single flat $(\exists\ \mathcal{M}^S, \mathcal{M}^T, \ldots \quad :: \quad \ldots)$ wrapping the conjunction of defining fragments and the kind-specific clauses. The implementation uses the flat form, reducing the size of emitted Boogie and avoiding the additional SMT quantifier nesting that the recursive scheme would introduce.

## IV. Specification Inference

### A. How it Works

MVP's *specification inference* [10] based on weakest-precondition (WP) analysis [17], [18] is made feasible and modular via behavioral predicates. WP inference takes arbitrary Move code and, for each function, computes conditions that are both necessary and sufficient for the implementation's behavior. Spec inference uses behavioral predicates to express the effect of calling out to another function, whether through a function value or a direct call. It reverses the translation seen for $invoke_\tau$ in Sec. III-C, introducing the behavioral predicates into the WP to express the call. Without this modular abstraction, inferred specs would be impractically large.

A full description of MVP's WP analysis is out of scope of this paper. What is important is that, besides the notorious loop invariant problem for WP, specification inference for Move is precise and complete. This serves two purposes for this work, demonstrated in the following subsections: reducing the specification burden for compliance checking of function values (Sec. IV-B), and providing a validation pipeline that confirms the encoding is internally consistent (Sec. IV-C).

### B. Compliance Checking

Recall from Sec. II-B that the `Pool` struct carries a `pricing` closure constrained by a no-abort invariant: the pricing function must not abort for any inputs in any state. Two concrete pricing implementations were shown: `product`, which is compliant (it never aborts), and `with_fee`, which is not (it aborts when the owner's `Fee` resource is absent). Spec inference makes this compliance distinction machine-readable without any manual annotation.

Given the implementation of `with_fee`, WP inference easily derives the aborts conditions (we leave out the ensures, which it also determines):

```
spec with_fee {
  aborts_if [inferred] !∃<Fee>(owner);
  aborts_if [inferred] Fee[owner].bps > 10000;
  ...
}
```

Similarly, for the lambda example in Sec. II-C, we were required to attach a spec to a trivial lambda expression of the form `|x| x = 0`. Obviously, inference can help to avoid this. We are currently investigating relaxing the requirement of spec blocks for compliance-checked functions in those cases where they can be reliably inferred (no recursion and no loops); however, this is orthogonal to the work in this paper.

### C. Validation Pipeline and Test Coverage

Inferred conditions are labeled `[inferred]` and feed directly back into MVP for verification, creating a pipeline: code $\rightsquigarrow$ code+specs $\rightsquigarrow$ VC, with verification expected to succeed. This pipeline acts as a *differential-testing oracle* for the extension described in this paper: inference and verification are independent implementations which meet only at the semantics of behavioral predicates and state labels, so a failure indicates a defect in one of them — conditions inferred unsoundly or not tightly, or axioms too weak to discharge them. Tab. I maps the categories of the inference test suite [19] — currently 27 cases with over 550 inferred conditions — to the HOF patterns introduced in this paper. All categories pass end-to-end, evidence that the encoding is internally consistent and that the axioms are strong enough to discharge *tight* specifications: usable, not merely sound. Mutual recursion deserves a note: inference breaks the specification cycle by phrasing each function's spec in terms of the other's behavioral predicate, which verification resolves against the axioms of Sec. III-D.

### D. Empirical Status

The extension described in this paper is implemented in the production MVP shipped with the Aptos toolchain; the paper

TABLE I: Inference test categories that exercise behavioral predicates.

| Category | Representative cases |
|---|---|
| Direct calls | Single and chained calls to a spec'd helper; inferred specs use `ensures_of` and `aborts_of` to abstract over the callee's effect rather than inlining its body. |
| Mutual recursion | Mutually recursive `is_even`/`is_odd`; inference breaks the cycle by phrasing each spec in terms of the other's behavioral predicate. |
| Higher-order parameters | A function that takes a closure and applies it; the inferred spec abstracts the closure via `ensures_of`/`aborts_of` on the parameter, with no commitment to a particular implementation. |
| Stored-closure dispatch (enum) | Calculator with an enum variant carrying a closure field; dispatch through the variant is summarized by behavioral predicates on that field. |
| Stored-closure dispatch (struct field) | Vault with a `Strategy` closure stored in a struct field on Aptos framework types; inference emits `modifies_of` for the field together with the field-form behavioral predicates. |
| Pre/post state labels | `move_to`, `move_from`, mutable resource indexing; inferred specs distinguish `@pre` from post-state on the memory the predicate observes. |
| Intermediate state labels | Sequenced state-modifying calls (e.g. remove-then-publish); inferred specs introduce existentially quantified intermediate labels and pin them to defining behavioral predicates. |

describes Move language version 2.4 and MVP as of the aptos-core commit pinned in [11], [19]. The closure portion of the prover's test suite comprises 19 test modules with about 240 verified functions and 66 expected-error diagnoses, including the examples of this paper; together with the inference suite, these run in continuous integration on every change to the prover. The tests are required to be stable across changes to the prover and underlying tools like Z3, and complete in under 5 minutes on standard GitHub runners, with a default timeout of 40 seconds per verification condition. A benchmark over a larger corpus of higher-order Move contracts does not yet exist: higher-order Move shipped in late 2025, and a corpus is only now forming, partly in closed-source projects.

## V. Related Work and Conclusion

### A. Related work

Higher-order programs with side effects have been a long-standing concern in program verification. Our approach to behavioral predicates descends from the relational tradition of VDM [20], Z [21], [22], and the B method [23], in which operations are specified by relations between pre- and post-states; state labels generalize this to intermediate snapshots and make the relational flavor composable.

Among contemporary verifiers for imperative code with higher-order functions, Verus [24] offers function-value specifications via *spec closures* and phantom ghost state; the approach is state-passing, with state threaded as an extra argument to specifications rather than quantified over directly. Our treatment is more directly relational: state labels denote memory snapshots and are bound by quantifiers in the surface syntax, avoiding the encoding cost of an explicit state argument and matching the style in which Move developers already write pre-/postcondition specifications. Also for Rust, Wolff et al. [25] verify closures in the Prusti verifier via *call descriptions*, which relate a closure call's arguments and results to its captured state within separation logic; our behavioral predicates play a similar role, but are relational over labeled memory snapshots, with Move's alias-free references and resource-separated memory removing the need for separation-logic machinery.

F* [26] likewise verifies higher-order, effectful code, internalizing the WP calculus in its type system via Dijkstra monads [27]. Both systems support pre-/postcondition specifications, quantifiers, behavioral abstraction over function values, and structured proof guidance — in Move via `proof` blocks, `lemma` declarations, and `calc`, `split`, `apply` statements. The treatment of memory differs: F* optionally supports separation logic (via the Steel and Pulse frameworks) to reason about heap separation within the logic, whereas Move uses traditional frame conditions via modifies declarations, and its resource and reference model separates memory syntactically at compile time — separation need not be reasoned about by the SMT solver.

Dafny [28] is a close cousin of MVP: it has a similar specification model of pre-/postconditions, higher-order functions, and behavioral predicates. However, Dafny does not allow function values to modify state. This would be a severe restriction for Move, since it is a common usage pattern. Moreover, Move's resource-separated global memory and alias-free mutable reference model is not available in Dafny, as is also the case in F*. While both languages support state labels, Move state labels as described here can be used in abstract specifications via universal and existential quantification.

Lean 4 [29] is a dependently-typed interactive theorem prover [30], [31] supporting higher-order functions, with tactic-driven, kernel-checked verification. Unlike MVP, Lean does not work directly on a programming language but requires the user to encode it in its meta logic, typically modeling stateful systems via monads. MVP, in contrast, integrates directly with Move and aims at automated SMT-based verification, with optional proof scripts.

In the smart-contract setting, dynamic dispatch is a central challenge for Solidity verifiers. Certora [32] handles dispatch on Solidity by enumerating the possible callees of a virtual call (derived from the contracts in scope) and generating a case-split proof obligation per callee; summaries can be supplied by the user to abstract external calls. Other SMT-based Solidity tools [33], [34], [35] handle dynamic dispatch through similar case-splitting, with varying degrees of abstraction. MVP is more general: a call site's dispatch is explicit in the closure's

datatype, and our encoding lets the prover automatically fall back from case-split (when the concrete function is known) to behavioral-predicate reasoning (when only the function type is known), without the user having to structure the specification differently.

### B. Future Work

The current encoding enumerates all closure variants reaching a given call site; scaling to programs where the variant set is large or open (e.g. dynamically loaded modules) remains future work. State labels are currently individually bound; extending quantification to *sequences* of states ($\forall S_0..S_n$) would allow direct contracts for fold-like iteration, where today the higher-order function is verified once against a summary spec (Sec. II-C) — the flat existential encoding of Sec. III-E already handles chains of fixed length and would extend naturally. Proof automation for non-linear arithmetic arising in pricing formulas and similar domains currently relies on user-supplied proof hints; integrating portfolio arithmetic decision procedures is a natural direction. The spec inference pipeline produces behavioral-predicate-based specifications from arbitrary code; extending it to synthesize loop invariants involving function values automatically would reduce the remaining manual annotation burden.

### C. Conclusion

We extended MVP to Move 2's higher-order functions and dynamic dispatch. The extension rests on two language additions — behavioral predicates and state labels — and on a Boogie encoding that discriminates closure invocations on the function-value variant reaching the call site. The design preserves MVP's production-oriented cost model: routine, predictable, counterexample-driven verification, also in the presence of dynamic dispatch. We further extended MVP's specification inference to closures and dynamic dispatch, turning compliance of a function value with a behavioral invariant into an automated verification problem.

## References


[1] D. L. Dill, W. Grieskamp, J. Park, S. Qadeer, M. Xu, and J. E. Zhong, "Fast and reliable formal verification of smart contracts with the move prover," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2022)*, ser. Lecture Notes in Computer Science, vol. 13243. Springer, 2022, pp. 183–200.

[2] S. Blackshear, D. L. Dill, S. Qadeer, C. W. Barrett, J. C. Mitchell, O. Padon, and Y. Zohar, "Resources: A safe language abstraction for money," 2020. [Online]. Available: https://arxiv.org/abs/2004.05106

[3] Aptos Labs, "The Move on Aptos Book," https://aptos-labs.github.io/move-book/, 2023, accessed: 2026-05-09.

[4] J. Park, T. Zhang, W. Grieskamp, M. Xu, G. D. Giacomo, K. Chen, Y. Lu, and R. Chen, "Securing Aptos Framework with Formal Verification," in *5th International Workshop on Formal Methods for Blockchains (FMBC 2024)*, ser. Open Access Series in Informatics (OASIcs), vol. 118. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024, pp. 9:1–9:16.

[5] Aptos Labs, "The Aptos Framework Book," https://aptos-labs.github.io/framework-book/, 2023, accessed: 2026-05-09.

[6] W. Grieskamp, "Move goes higher order," https://medium.com/aptoslabs/move-goes-higher-order-cdf9688a4919, Oct. 2025, aptos Labs blog.

[7] K. R. M. Leino and C. Pit-Claudel, "Trigger Selection Strategies to Stabilize Program Verifiers," in *Proceedings of the 28th International Conference on Computer Aided Verification, Part I*. Springer, 2016, pp. 361–381.

[8] C. Barrett, A. Stump, and C. Tinelli, "The SMT-LIB Standard: Version 2.0," in *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*, A. Gupta and D. Kroening, Eds., 2010.

[9] L. M. de Moura and N. Bjørner, "Z3: an efficient SMT solver," in *TACAS*, ser. Lecture Notes in Computer Science, vol. 4963. Springer, 2008, pp. 337–340.

[10] W. Grieskamp, T. Zhang, and V. Kashyap, "Combining mechanical and agentic specification inference for Move," *CoRR*, vol. abs/2605.10005, 2026. [Online]. Available: https://arxiv.org/abs/2605.10005

[11] Aptos Labs, "Higher-Order Paper Examples," https://github.com/aptos-labs/aptos-core/tree/28f82080e3/third_party/move/move-prover/doc/higher-order-paper-26/examples, 2026, commit `28f82080e3`, accessed 2026-07-28.

[12] J. E. Zhong, K. Cheang, S. Qadeer, W. Grieskamp, S. Blackshear, J. Park, Y. Zohar, C. Barrett, and D. L. Dill, "The Move Prover," in *Computer Aided Verification*, S. K. Lahiri and C. Wang, Eds. Springer International Publishing, 2020, pp. 137–150.

[13] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino, "Boogie: A modular reusable verifier for object-oriented programs," in *International Symposium on Formal Methods for Components and Objects*. Springer, 2005, pp. 364–387.

[14] K. R. M. Leino, "This is boogie 2," 2008, manuscript KRML 178. [Online]. Available: https://www.microsoft.com/en-us/research/publication/this-is-boogie-2-2/

[15] H. Adams, N. Zinsmeister, and D. Robinson, "Uniswap v2 core," https://uniswap.org/whitepaper.pdf, Mar. 2020, accessed: 2026-04-21.

[16] K. R. M. Leino and P. Rümmer, "A polymorphic intermediate verification language: Design and logical encoding," in *TACAS*, J. Esparza and R. Majumdar, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 312–327.

[17] E. W. Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs," *Communications of the ACM*, vol. 18, no. 8, pp. 453–457, 1975.

[18] M. Barnett and K. R. M. Leino, "Weakest-precondition of unstructured programs," in *Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. New York, NY, USA: Association for Computing Machinery, 2005, pp. 82—87.

[19] Aptos Labs, "Move Prover Spec Inference Test Suite," https://github.com/aptos-labs/aptos-core/tree/28f82080e3/third_party/move/move-prover/tests/inference, 2026, commit `28f82080e3`, accessed 2026-07-28.

[20] C. B. Jones, *Systematic Software Development Using VDM*, 2nd ed. Prentice-Hall, 1990.

[21] J. M. Spivey, *The Z Notation: A Reference Manual*, 2nd ed. Prentice-Hall, 1992.

[22] W. Grieskamp, "A computational model for Z based on concurrent constraint resolution," in *ZB 2000: Formal Specification and Development in Z and B*, ser. Lecture Notes in Computer Science, J. P. Bowen, S. Dunne, A. Galloway, and S. King, Eds., vol. 1878. Springer, 2000, pp. 414–432.
[23] J.-R. Abrial, *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.
[24] A. Lattuada, T. Hance, C. Cho, M. Brun, I. Subasinghe, Y. Zhou, J. Howell, B. Parno, and C. Hawblitzel, "Verus: Verifying rust programs using linear ghost types," in *Proc. ACM Program. Lang.*, vol. 7, no. OOPSLA1, 2023, pp. 286–315.
[25] F. Wolff, A. Bílý, C. Matheja, P. Müller, and A. J. Summers, "Modular specification and verification of closures in Rust," *PACMPL*, vol. 5, no. OOPSLA, pp. 1–29, 2021.
[26] N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J.-K. Zinzindohoué, and S. Zanella-Béguelin, "Dependent types and multi-monadic effects in F*," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 2016, pp. 256–270.
[27] K. Maillard, D. Ahman, R. Atkey, G. Martínez, C. Hritcu, E. Rivas, and E. Tanter, "Dijkstra monads for all," in *24th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2019. [Online]. Available: https://arxiv.org/abs/1903.01237
[28] K. M. Leino, "Accessible software verification with dafny," *IEEE Software*, vol. 34, no. 06, pp. 94–97, nov 2017.
[29] L. de Moura and S. Ullrich, "The Lean 4 theorem prover and programming language," in *Automated Deduction – CADE 28*, ser. Lecture Notes in Computer Science, A. Platzer and G. Sutcliffe, Eds., vol. 12699. Springer, 2021, pp. 625–635.
[30] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, ser. Lecture Notes in Computer Science. Springer, 2002, vol. 2283.
[31] Y. Bertot and P. Castéran, *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*, ser. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.
[32] Certora, "The Certora Prover," https://www.certora.com/, 2024.
[33] L. Alt and C. Reitwießner, "SMT-Based Verification of Solidity Smart Contracts," in *ISoLA (4)*, ser. Lecture Notes in Computer Science, vol. 11247. Springer, 2018, pp. 376–388.
[34] Á. Hajdu and D. Jovanovic, "solc-verify: A modular verifier for Solidity smart contracts," *CoRR*, vol. abs/1907.04262, 2019.
[35] Á. Hajdu and D. Jovanovic, "SMT-Friendly Formalization of the Solidity Memory Model," in *ESOP*, ser. Lecture Notes in Computer Science, vol. 12075. Springer, 2020, pp. 224–250.