ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Verification of Programs Written in Libra's Move Language

Master's Thesis

Constantin Müller

September 02, 2020

Supervisors: Prof. Dr. Peter Müller, Vytautas Astrauskas, Marco Eilers, Federico Poli
Department of Computer Science, ETH Zürich

Abstract

In recent years, blockchain-based cryptocurrencies have exploded in popularity. At the same time, research into blockchain-based systems has led to huge advances, such as decentralised applications called smart contracts. The Libra project aims to build a new, pervasive financial infrastructure based on smart contracts. In order to avoid some of the mistakes that previous smart contract programming languages have made, the Libra developers decided to develop a new language called Move.

Since smart contracts deal with potentially huge sums of money, making software bugs very expensive, it is critical to ensure that smart contracts behave in the way that is intended by their developer. Formal verification intends to prove the correctness of a program, given a specification of the program's behaviour. In this thesis, we design a specification language for Move and build a formal verifier for Move programs based on the Viper infrastructure.

The goal of our verifier is to leverage Viper's permission-based model to achieve modular verification. We explore to what extent we can use Move's type system for framing, and exploit the guarantees provided by Move's type system in our encoding to Viper. The resulting verifier supports a significant subset of the Move language and is able to prove non-trivial properties of complex Move code, even as performance is currently lacking.

Acknowledgements

First and foremost, I would like to thank my supervisors Federico Poli, Marco Eilers and Vytautas Astrauskas for all the stimulating discussions we have had and all the time they have put into this project. I could not have asked for a better group of supervisors. I would also like to thank Prof. Dr. Müller for giving me the opportunity to work on such an exciting topic in such an inspiring group.

I am also grateful to my parents for their endless support and encouragement. Last but not least, I would like to thank my sister Katharina, my girlfriend Nathalie, Luka and Will for being wonderful and exceptional people who have supported me throughout this thesis and my entire life.

Contents

Chapter 1

Introduction

1.1 Libra, Move and Verification

Libra [1] is a blockchain-based cryptocurrency initiated by Facebook that aims to underpin a new global financial infrastructure. The idea behind a blockchain is to keep a growing list of transactions that are linked cryptographically, making past transactions immutable and permanent. These transactions allow the transfer of resources, e.g., sending money from one person to another. Smart contracts are applications that perform transactions on the blockchain based on a set of rules that are encoded in a blockchain-specific programming language. These applications can, for example, provide banking and insurance services, allowing the blockchain to be the basis of an extensive financial infrastructure.

Due to the intended scale and monetary nature of the Libra blockchain, the correctness of smart contracts is of the utmost importance. In the past, bugs in smart contracts for Ethereum, the most-widely used blockchain system with support for smart contracts, have caused financial damages in the hundreds of millions US dollars [8]. Formal verification is widely considered the gold standard for ensuring the correctness and safety of code. Formal verification evaluates code with respect to a user-provided specification, determining whether the program's behaviour matches the properties specified in every situation. For many programming languages, formal verification is difficult as it is hard to reason about invalid references, dynamic call sites, concurrent programs, etc. In many cases, writing a program specification is only possible for verification specialists. The Libra developers, however, envisage a system in which all smart contracts are verifiable and annotated with specifications, enabling the formal verification of all transactions made on the blockchain.

With this goal in mind, the Libra developers are currently working on developing the *Move* [2] language in which all programs for the Libra blockchain

```
1  struct S {
2      field: u64
3  }
4
5  fun set_zero(s: &mut S) {
6      s.field = 0;
7      no_modification(&s);
8  }
9
10 fun no_modification(&s: S) { /* ... */ }
```

Figure 1.1: The Frame Problem

will be written. The Move language is designed to eliminate common bugs that have plagued other smart contract languages while also supporting relatively straightforward formal verification. In fact, during our work on this thesis and in parallel to the development of the Move language, the Libra developers have been developing their own experimental verifier, called the *Move Prover* [9], in order to verify the Move standard library. The Move Prover translates specifications and Move bytecode to *Boogie* [5], an intermediate verification language.

1.2 Viper

Viper [7] is another intermediate verification language that internally uses Boogie and has been the basis of a number of verification front-ends. One of the key differences between Viper and Boogie is that Viper supports *permission-based* reasoning in the style of separation logic [4]. Prusti [3], a verification front-end for Rust, in particular, has been able use Viper's permission-based reasoning to encode properties of the Rust language in a natural way, supporting a method of verification that uses specifications purely on a level of the Rust language itself.

The core goal of this thesis is to show how the Move language can be encoded to Viper and that non-trivial properties about Move programs can be proved in this way. We will also compare our work on this verifier to the Boogie-based Move Prover throughout.

1.3 The Frame Problem

One of the current restrictions of the Move Prover is the lack of framing information during verification. Without focussing on the exact syntax and semantics of Move programs, consider the code in Figure 1.1. When specifying a postcondition for the `set_zero` function, we would like to be able to assert `s.field == 0`. Knowing that `no_modification` takes an immutable reference to `s` is sufficient for us to guarantee that `s.field` remains unchanged

after its assignment in line 6. When verifying the `set_zero` function against our `s.field == 0` postcondition we have two options:

- Either we inline the `no_modification` function and run our verification on its potentially large content (including calls made)
- or we use a specification of the `no_modification` to reason about the changes it makes.

The Move Prover currently applies live variables analysis to decide which option to use and will inline functions that may affect the state of the caller procedure. This verification process is, therefore, at the moment in part non-modular and requires reasoning about function behaviour and possibly the execution of a function body every time a function is called. This may lead to an explosion of verification efforts as the number of functions and calls between them increase.

The latter option requires a specification that describes the function's behaviour and, in particular, information about which non-local variables may be changed. This information is known as the function *frame*.

1.4 Contributions

In this thesis, we make three core contributions:

1. We define a specification language for Move programs that is able to describe the changes made by a Move program to the global state. The specification language also completes framing information and is able to describe non-trivial global properties.
2. We describe in detail how to encode Move programs and program specifications to Viper. We also describe how we encode Move's type system guarantees to Viper in order to simplify verification.
3. We implement the encoding in a prototype verifier for Move programs.
4. We evaluate the prototype verifier with respect to a number of Move programs and compare it to the Move Prover.

We will begin by describing the Move language in more detail in the following chapter. In subsequent chapters, we will describe each one of the core contributions.

Chapter 2

Background on Move

In this chapter, we will summarise the main features of the Move language. More details on the Move language are provided in [2] and in the open-source Libra repository[1].

2.1 Global State

The global state of the Libra blockchain is determined by a mapping from accounts to resources.[2] Libra supports an arbitrary number of accounts, which are somewhat similar to Ethereum wallets. Every account has a unique address associated with it. An account can hold an arbitrary number of *resources*, though only ever one resource of a kind. Resources are one of the novelties of the Move language and their semantics will be described in more detail in Section 2.3. Resources represent assets, such as currency, permissions (e.g. voting rights) and ownership (e.g. property ownership). Figure 2.1 shows an example of what the global state might look like: every address may hold some number of resources and every resource may contain additional information in *fields*.

2.2 Transaction Scripts and Modules

Every resource type must be defined as part of a *module*. Figure 2.2 shows an example definition of the `Coin` resource type in the `Currency` module. Modules are always published by an account. The account's address is part of the module's identifier, as in `0x1234::Currency`[3]. In procedures outside of their module, resources act as *linear types*, which means that they may only be moved from one procedure to another, but never created, destroyed, modi-

[1] https://github.com/libra/

[2] For our purposes, the mechanics of the blockchain's blocks are not important.

[3] Addresses are shown as hexadecimal numbers in Libra.

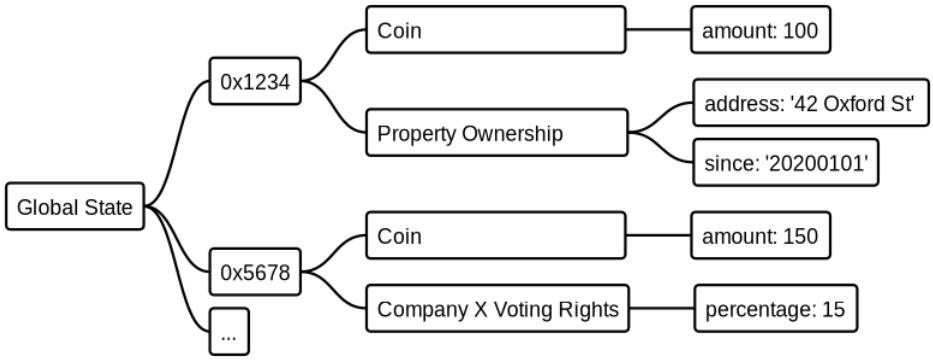

Figure 2.1: An example global state of Libra

```
1  module Currency {
2      resource struct Coin {
3          amount: u64
4      }
5  }
```

Figure 2.2: Definition of a resource type

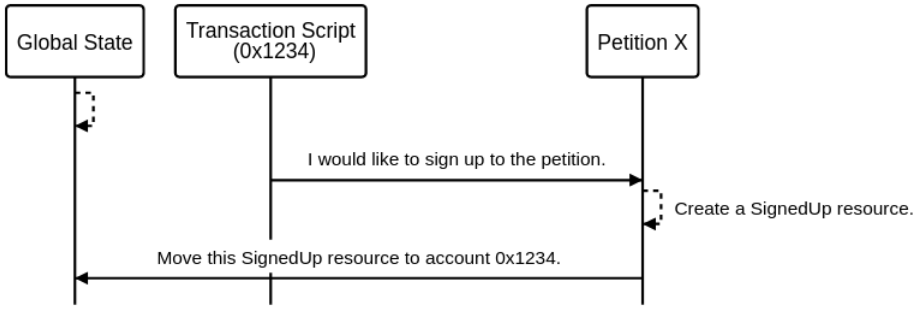

Figure 2.3: Illustration of a sample transaction.

fied or duplicated. In Section 2.3.1 we will see how useful this property is for verification. *Module procedures*, in contrast, may create, modify and destroy resources without restrictions. These operations are, of course, necessary to modify the global state.

Transactions on the Libra blockchain, which modify the global state, are initiated by *transaction scripts*, which are single-use Move programs[4] that make calls to public module procedures. These module procedures then make calls to other procedures and/or execute operations on resources. Figure 2.3 shows a small sample transaction. The *transaction sender*, in this case with address `0x1234`, would like to sign up to a petition. To do that, the trans-

[4]They only contain a single `main` function.

```
1 fun distribute(coin: Coin, payee1: address, payee2: address) {
2     let coin_amount = Currency::value(&coin);
3     let halved_amount = coin_amount / 2;
4     let coin2 = Currency::split(&mut coin, halved_amount);
5     Currency::deposit(coin);
6     // Currency::deposit(coin2);
7 }
```

Figure 2.4: The `distribute` function fails because it violates the linearity of resources.

action script makes a call to a procedure in the `Petition` module, which is automatically permitted to create a `SignedUp` resource. It then moves this resource to the account with address `0x1234`.

There are thus two kinds of Move programs, modules, which define resource types as well as procedures allowed to perform operations on resources, and transaction scripts, which make calls to module procedures to change the global state.

Note that unlike Ethereum smart contracts, Move modules do not have their own stored state. The state is fully determined by the resources held by accounts. While it is possible for a module developer to use one particular account to store some information, resources encourage developers to distribute the state amongst many accounts. As we will see in future sections, this design often makes reasoning more local and thus simpler.

2.3 Resource Syntax and Semantics

2.3.1 Linearity

As mentioned in the previous section, resources outside procedures of the module that defines their type act as linear types. Figure 2.4 shows a `distribute` procedure that is supposed to take a `Coin` resource (defined in the `Currency` module) and split it equally between two payees. The `Currency` `split` procedure mutates the `Coin` and returns a second `Coin`, both of which now have the same value. Suppose now that the developer forgets to deposit one of the coins (line 6). In this case, the linearity of resources would be violated, as a resource would be implicitly destroyed. Therefore, the compiler[5] would reject this program, preventing the kind of bug where the developer "forgets" about assets.[6]

The linearity of resources has a profound implication for verification. It makes resources immutable outside their module which means that prop-

[5] More precisely, the *bytecode verifier*, which also performs a number of other checks.

[6] On the Ethereum platform, these have led to very expensive bugs. The *Flint* [6] smart-contract language has also introduced *Assets*, which follow linear semantics, to prevent those.

```
module M {
    resource struct T {
        value: u64
    }

    public fun pack_resource(): T {
        let t: T = T {
            value: 42
        };
        t
    }
}
```

Figure 2.5: Resource packing

```
module M {
    resource struct T {
        value: u64
    }

    public fun unpack_resource(t: T): u64 {
        let T { value } = t;
        value
    }
}
```

Figure 2.6: Resource unpacking

```
module Petition {
    resource struct SignedUp { }

    public fun sign_up(transaction_sender: &signer) {
        let s: SignedUp = SignedUp { };
        move_to<SignedUp>(transaction_sender, s);
    }
}
```

Figure 2.7: Resource packing and moving

erty invariants on resources that hold inside their module are global invariants, as without modification invariants cannot be broken by other modules. In Chapter 3 we will see how we can exploit this in our specifications.

2.3.2 Packing and Unpacking

Module procedures can freely create resources of types defined by their module. In Figure 2.5 we see an example of a procedure that *packs* (creates "out of thin air") a resource of type `T` and then returns it. Similarly, a resource may be *unpacked* into its fields, which destroys the resource. An example of this is shown in Figure 2.6.

```
1  module Petition {
2      use 0x1::Signer;
3
4      // ...
5
6      public fun deregister(transaction_sender: &signer) acquires SignedUp
           {
7          let sender_address = Signer::address_of(transaction_sender);
8          let s = move_from<SignedUp>(sender_address); // cannot discard
               here
9          let SignedUp {} = s; // need to first unpack to discard
10     }
11 }
```

Figure 2.8: Syntax of `move_from` and `moves_from`

2.3.3 Moving

Recall that in our petition transaction as shown Figure 2.3, the `Petition` module needs to provide a procedure to `sign_up`, which creates a `SignedUp` resource and moves it to the transaction sender. In Figure 2.7, we can see the Move code for this.

Note that procedures are *not* allowed to move resources to arbitrary accounts. The `move_to` built-in function requires an argument of type `&signer` , a signer of the transaction, which will generally refer to the transaction *sender address*. The transaction developer needs to pass this argument to the module procedure to indicate that she is happy for the sender address to receive resources.

Recall that every account can hold at most one resource of a kind. Moving another resource to the account would therefore lead to a run-time error. Move deals with this kind of run-time error by *aborting* the transaction, reverting all the changes that have been made to the global state. Aborting is nothing unusual for transactions, it is the default way of dealing with unintended input states.

In addition to moving resources to accounts, Move also supports moving resources from accounts into a procedure[7], as shown in Figure 2.8. Here we deregister the sender by moving the `SignedUp` resource into the procedure and then discarding it. For the `move_from` operation, no `&signer` variable is required, the procedure is privileged to remove resources from any account where such a resource exists. We can, however, obtain the address of a `&signer` using the special `Signer` module[8] (line 7) that is imported in line 2.

The Move language requires the resource types being moved from an ad-

[7] Of course only if the resource type is defined in the procedure module.

[8] The `Signer` module is part of the standard library. All standard library modules are stored under address `0x1`.

```
1  module M {
2      resource struct T { value: u64 }
3
4      public fun set_value(t_ref: &mut T, new_value: u64) {
5          let field_ref = &mut t_ref.value;
6          *field_ref = new_value;
7
8          // equivalent to
9
10         t_ref.value = new_value;
11     }
12 }
```

Figure 2.9: Syntax of resource modification

dress to be specified in the procedure's `acquires` list (line 6). The `acquires` list is used to establish certain type system guarantees. In this case, the compiler uses it to prevent us from modifying a resource that may have been removed. The `acquires` list will be discussed in more detail in Section 2.4.

2.3.4 Modification

Now that we have seen how to pack, unpack and move resources in Move, we can in theory write module and transaction scripts that achieve any desired change in the global state. To make the language more practical, however, it should be possible to modify a resource without first having to move, then unpack, pack and lastly move it again. To this end, Move supports mutable references to resources, which may either live under an account address or have been created in the current transaction. The `set_value` in Figure 2.9 shows two ways of modifying a resource field.

As mentioned above, resource references may refer to resources that live in the current transaction or resources that live in the global state. Figure 2.10 shows what these two options look like in practice.

In the case of the `get_from_global_and_modify` procedure, there is no guarantee that there is a resource of type `T` at address `addr`. Similar to the semantics of `move_to` and `move_from`, the `borrow_global_mut` instruction will cause the transaction to abort if no such resource exists.

2.4 Type System

Now that we understand the core semantics of Move, let us take a closer look at its type system. A summary of the simplified version of Move's type system that we will be using is shown in Figure 2.11.

```
module M {
    resource struct T { value: u64 }

    // ...

    public fun pack_and_modify() {
        let t = T {
            value: 0
        };
        modify(&mut t);
        // need to consume resource t
    }

    public fun get_from_global_and_modify(addr: address) acquires T {
        let t_ref: &mut T = borrow_global_mut<T>(addr);
        modify(t_ref);
        // the value of the resource T at address 'addr' has been
            modified
    }
}
```

Figure 2.10: Two ways of obtaining a mutable reference to a resource

$$
\begin{array}{ll}
\texttt{PrimitiveType}: & \texttt{address} \cup \texttt{bool} \cup \texttt{u64} \cup \texttt{signer} \\
\texttt{Resource}: & \texttt{FieldName} \rightarrow \texttt{NonRefType} \\
\texttt{Vector}: & \texttt{NonRefType[]} \\
\texttt{NonRefType}: & \texttt{Resource} \cup \texttt{PrimitiveType} \cup \texttt{Vector} \\
\texttt{Type}: & \texttt{NonRefType} \cup \texttt{\&NonRefType} \cup \texttt{\&mut NonRefType}
\end{array}
$$

Figure 2.11: The Move Type System (Simplified)

2.4.1 Primitive Types

Move supports six primitive types:

- `bool`
- `u8` (unsigned 8-bit integer)
- `u64` (unsigned 64-bit integer)
- `u128` (unsigned 128-bit integer)
- `address` (an account address)
- `signer` (wraps the transaction sender address)

In this thesis, we will not treat `u8`, `u64` and `u128` separately and only consider `u64`.

2.4.2 Vectors and Resources

As we have seen in our examples, resources can contain zero or more fields of primitive types. Resources may, however, also contain other resources,

though a resource type cannot contain itself and there must not be any resource dependency cycles. A contained resource may either be defined in the current module or in another. This places resources in a kind of *escrow*, as the inner resource cannot be accessed without access to the outer one (to which only module procedures are privileged).

In addition to resources and primitive values, resources can also contain `vectors`. Vectors are built into Move, though vector operations are defined in the standard library `Vector` module. Vectors, in turn, contain elements of a generic type, which may be a vector, a resource or a primitive type. Move also supports non-resource structs, which contain non-resource fields. Since non-resource structs are not very interesting for verification, we ignore them in this work. We also ignore tuple types in this work.

2.4.3 References and Borrow-Checking

References

As shown in the previous section, Move supports references to resources. Move also supports references to resource fields and arbitrary local variables, however, references to references are not supported. Move references may either be mutable or immutable.

Ownership and Borrow-Checking

Move's type system also contains an ownership system similar to Rust's. In short, every variable has exactly one owner at any point in time, although the owner may change. When we define a new variable `var` in a procedure, initially the procedure is the owner of that variable. When we pass this variable as an argument to another procedure, as in `foo(var)`, ownership is transferred and we can no longer use `var` in our procedure. If we would like to continue using it after the call, we shouldn't pass the variable itself but a reference to it, `foo(&var)` or `foo(&mut var)`, depending on whether the variable may be mutated or not. We say that `var` is *borrowed* by `foo`.

One of the core guarantees made by Move's type system is that at any point in time a *mutable* reference to a variable is the only reference to the variable. In other words, mutable references cannot alias other references. To ensure this property, Move has a borrow-checker that ensures that a variable is not borrowed in multiple places at the same time if one of those borrows is mutable.

The `acquires` List

In Figure 2.10 we saw that we can use the built-in `borrow_global_mut` operation to obtain a mutable reference to a resource stored in an account. For

```
1  fun immutable_borrow() acquires T {
2      let t_ref = borrow_global<T>(0x1234); // immutable reference
3      mutable_borrow(); // changes a value behind the reference
4      // use t_ref
5  }
6
7  fun mutable_borrow() acquires T {
8      let t_ref = borrow_global_mut<T>(0x1234);
9      t_ref.value = 42;
10 }
```

Figure 2.12: Invalid due to aliasing borrows

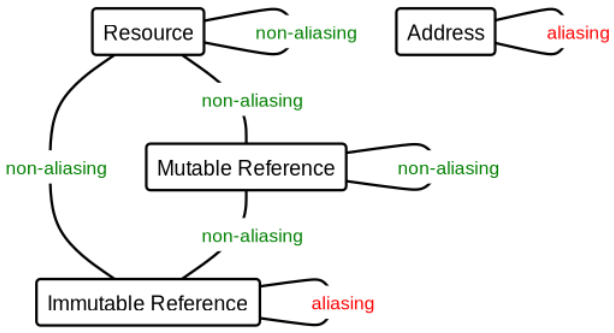

Figure 2.13: Aliasing in Move

these references the same guarantees need to apply. Consider now the code in Figure 2.12. The `immutable_borrow` procedure acquires an immutable reference to the `T` resource at address `0x1234`. The non-aliasing property guarantees that while the immutable reference is alive, i.e. before its last usage, there is no other mutable reference to the resource or one of its fields. For the borrow-checker to find out that the call to `mutable_borrow` is invalid, it would have to go through the procedure's code including other procedures called by this. This process' complexity could explode quickly and slow down compilation.

As a remedy, Move requires the developer to annotate which global resources are borrowed or moved into the procedure using the `acquires` list (line 1, 7). A procedure that has a live reference to a global resource must not call a procedure that acquires the same resource type. We will later use a similar specification to complete the frame information required to know what global resources may be modified.

Aliasing Summary

Thus far we have considered aliasing between resources and references. Later on we will see that it is also important for our specifications to reason about the aliasing of addresses where the type system makes no guarantees.

```
1  fun move_to_no_abort(transaction_sender: &signer) {
2      let addr = Signer::address_of(transaction_sender);
3
4      if (!exists<T>(addr)) {
5          move_to(transaction_sender, T {
6              value: 0
7          });
8      }
9  }
```

Figure 2.14: Using the `exists` operation

A summary of the aliasing rules can be seen in Figure 2.13.

2.5 Other Features

2.5.1 Supported Operations

Apart from the resource operations we have seen thus far, Move supports the standard arithmetic operations, branching with `if`-statements and loops. Like the Move Prover we do not reason about unbounded loops in this thesis. The last important Move operation that has not been mentioned so far is the `exists` operation. In past examples we have seen that a transaction will abort if we try to move a resource to an account that already holds such a resource.[9] To avoid this behaviour, we can branch on the existence of a resource at a location. How we can use the built-in `exists` operation is shown in Figure 2.14.

2.5.2 Verification-Friendliness

As mentioned in the introduction, Move attempts to avoid many of the mistakes other smart contract programming languages have made. While the introduction of resources is the most noticeable one, there are a number of other verification-friendly features that have not been mentioned yet:

- Call sites can be determined statically. Other smart contract programming languages support making dynamic calls where reasoning about the call target is very limited. The Move languages enforces this restriction by requiring all modules that another module imports to have been published previously. This also ensures that the module dependency graph is acyclic.
- When a Move procedure encounters a (well-typed) input it does not expect, it will typically cause the entire transaction to abort, resetting all the changes made to the global state. This provides clear semantics

[9] The same applies to borrowing/moving a resource from an account that does not hold one.

for how to handle unexpected inputs.

- Move performs run-time checks on integer overflows and underflows. This gives the developer a guarantee that if a transaction succeeds, there are no overflows or underflows.
- There is no concurrency and transactions can be considered to be atomic. Reasoning about concurrent programs is one of the hardest areas in verification. This is not something we have to worry about in Move programs.

Chapter 3

Specification of Move Programs

We will now analyse which properties of Move programs are relevant for verification and introduce a specification language to capture these properties. While this specification language was initially developed independently of the Move Prover's specification language, the two turned out to share many characteristics and features. In order to be able to use the Move Prover's infrastructure created by the Libra developers, we base the syntax of our specification language on the Move Prover's specification language. Whenever not indicated otherwise, the Move Prover's specification language supports the same syntax.

Recall that the state of the Libra blockchain is determined by a mapping from accounts to resources: every account may hold an arbitrary number of resources, though, never more than one resource of a kind. This state can only be modified by a *transaction script*, a Move program that calls procedures defined in *modules*.

The transaction script is typically initiated by one account.[1] Every account has a unique address which in this case we call the *sender address*. A transaction script may modify resources at the sender address *and also at other addresses*. In the most general case, the developer of a transaction script will want to verify the changes made to all resources at all addresses caused by the transaction. The developer of a *module* will need to write specifications that allows such verification of transactions. Additionally, the developer of a module may want to specify *global properties* of a module that help ensure correct behaviour of the procedures it defines. We will now look at the properties required to express the behaviour of transaction scripts and modules in more detail and define how we describe these in our specification language.

[1] The Move standard library contains procedures with multiple signers. In this chapter we will only consider transactions with a single signer.

```
1  public fun get_value(addr: address) acquires T {
2      let t_ref = borrow_global_mut<T>(addr);
3      add_address_queried(addr);
4      t_ref.value
5  }
6
7  spec fun get_value {
8      aborts_if !exists<T>(addr);
9  }
10
11 fun add_address_queried(addr: address) {
12     // add addr to some data structure
13 }
14
15 spec fun add_address_queried {
16     requires exists<T>(addr);
17 }
```

Figure 3.1: Abort condition and precondition

3.1 Specification of a Module Procedure

As with most formal specifications of functions, we would like to be able to specify Hoare triples for Move procedures. In this section, we will look at the specification of preconditions, postconditions and frame information.

3.1.1 Abort Conditions and Preconditions

Public module procedures may be called by any transaction script. Since we can generally make no guarantees about the state that the transaction is in at the time of calling the procedure, preconditions make little sense for public module procedures. Move's `abort` semantics, however, provide a way for us to specify conditions before the execution of the procedure required for the procedure to succeed. We introduce the `aborts_if` precondition that specifies a sufficient condition for the transaction to abort. If an `aborts_if` condition is not met, in a sense, no harm is done as all changes will be reverted.[2] These conditions may refer to the procedure's arguments or the global state *before the procedure call*.

Private Move procedures, in contrast, may have preconditions, as the writer of a module has full control over when those procedures are called. For such preconditions, we introduce the `requires` keyword that is followed by a necessary condition to call a procedure. Let us now look at an example specification of an `aborts_if` condition and a `requires` condition.

Figure 3.1 shows one public procedure, `get_value`, and one private procedure, `add_address_queried`. Note that we write the procedure annotations

[2] A failed transaction will nevertheless cost the transaction sender *gas*, which is why they may want to verify that their transaction will not abort.

```
1  public fun inc(x: &mut u64) {
2      *x = *x + 1;
3  }
4
5  spec fun inc {
6      aborts_if x == max_u64();
7  }
```

Figure 3.2: Overflow specification

inside a *specification block* indicated by `spec fun procedure_name`. In our example, the `add_address_queried` procedure is used keep track of all the addresses that have been queried, which may only make sense if a resource exists at that address. The precondition `requires exists<T>(addr)` required to make the procedure call in line 3 is established by the `borrow_global_mut` operation in line 2.

When specifying an `aborts_if` condition, we want to exclude certain states from our reasoning about the function's behaviour, which achieves the same goal as classical preconditions. Additionally, with `aborts_if` conditions, we can ensure that the developer specifies *all* `aborts_if` conditions required for the procedure to act as intended. We do this by verifying that the procedure aborts if and only if one or more of the conditions are met. This is a powerful mechanism to ensure that the developer has specified all relevant conditions required for the procedure to succeed. If an `aborts_if` condition is specified that does not lead to an abort or an abort is not covered by any such condition, our verification process will return an error.

We also use `aborts_if` conditions to reason about integer overflows (exceeding the maximum unsigned 64-bit value) and underflows (obtaining a value less then zero). To ensure that no overflow takes place for a `u64` variable, we verify that the procedure succeeds only if the maximum value is not exceeded. For this purpose, we introduce the `max_u64` function that returns the maximum unsigned 64-bit value. Figure 3.2 shows an example of this. Since all conditions sufficient for an abort must be specified, the developer also needs to specify an `aborts_if` condition for every possible overflow or underflow.

3.1.2 Postconditions

A Move procedure may make local changes (modify mutable references, return values) and global changes (move/modify account resources). In this section, we will look at how we can specify these changes. While some of these changes may not be well defined in all circumstances, e.g. modifying a resource at an address where none exists, a procedure's `aborts_if` conditions help the developer and the verifier ensure that the postconditions refer to well-defined resources.

```
1  module M {
2      resource struct T {
3          x: u64,
4          y: u64
5      }
6
7      public fun modify_x(t_ref: &mut T) {
8          t_ref.x = 42;
9      }
10
11     spec fun modify_x {
12         ensures t_ref.x == 42;
13         ensures t_ref.y == old(t_ref.y);
14     }
15 }
```

Figure 3.3: Specification of the changes made to a mutable reference

Mutable References

As we saw in Section 2.4.3, Move supports mutable references to local variables and global resources and fields. When a mutable reference is passed to a procedure, the developer should specify how this mutable reference has changed. We introduce the `ensures` keyword to specify a postcondition and use the `old` keyword to refer to the state before the procedure call. Figure 3.3 shows an example of what a postcondition specification look like.

Return Values

Next, let us specify postconditions on return values. We refer to a procedure's return values as `result` or `result_1`, `result_2`, etc., respectively, if a procedure returns a tuple of multiple values. Figure 3.4 shows an `Ownership` module with a resource type `T` containing a single field `percentage`. We now specify the behaviour of a function that splits a `T` resource into two with equal percentages by writing `ensures result_1.percentage == t.percentage / 2` and `result_2.percentage == t.percentage / 2` postconditions. This is not supported where the percentage is not even (an abort condition).

Account Resources

Now that we have considered changes to the local state, let us look at modifications of the global state. As we saw in Section 2.3.4, Move procedures are allowed to obtain mutable references to account resources where the type is defined in the procedure's module. We use `global<ResourceType>(account_address)` expressions in our specifications to refer to account resources. Figure 3.5 shows as an example a `Currency` module with a `Coin` resource that provides a `deposit` procedure, which ensures that the value of some `Coin` resource is added to a beneficiary's `Coin` resource. The postcondition of lines 15 and 16 refers to the global state of the resources before and

```
module Ownership {
    resource struct T {
        percentage: u64
    }

    public fun divide_equally(t: T): (T, T) {
        let T { percentage } = t; // unpack
        if (percentage % 2 != 0) {
            abort 100 // abort with abort number 100
        };
        let halved: u64 = percentage / 2;
        let t1 = T { // pack first result value
            percentage: halved
        };
        let t2 = T { // pack second result value
            percentage: halved
        };

        (t1, t2) // return
    }

    spec fun divide_equally {
        aborts_if t.percentage % 2 != 0;
        ensures result_1.percentage == t.percentage / 2;
        ensures result_2.percentage == t.percentage / 2;
    }
}
```

Figure 3.4: Specification of the return value of a procedure

```
module Currency {
    resource struct Coin {
        value: u64
    }

    public fun deposit(beneficiary: address, coin: Coin) acquires Coin {
        let Coin { value } = coin; // unpack
        let coin_ref = borrow_global_mut<Coin>(beneficiary); // borrow
            account resource
        coin_ref.value = coin_ref.value + value; // update value
    }

    spec fun deposit {
        aborts_if !exists<Coin>(beneficiary);
        aborts_if global<Coin>(beneficiary).value + coin.value > max_u64
            ();
        ensures global<Coin>(beneficiary).value ==
            old(global<Coin>(beneficiary).value) + coin.value;
    }
}
```

Figure 3.5: Specification of resource modification

```
1  module M {
2      resource struct T {
3          value: u64
4      }
5
6      fun dec(addr: address) {
7          if (exists<T>(addr)) {
8              let t_ref = borrow_global_mut<T>(addr);
9              t_ref.value = t_ref.value - 1;
10         };
11     }
12
13     spec fun dec {
14         ensures exists<T>(addr) ==> global<T>(addr).value ==
15             old(global<T>(addr).value) - 1;
16     }
17 }
```

Figure 3.6: Specification of conditional resource modification

after the procedure call.

Such postconditions may be conditional. Let us suppose we want to write a procedure that decrements the value of some resource but does not abort if the resource does not exist. In that case, our postcondition would depend on whether the resource exists, which we express as an implication as shown in Figure 3.6.

Resource Holders

The second modification of the global state we need to reason about is the moving of resource from or to accounts. Recall our `Petition` module from Chapter 2 that moves a `SignedUp` resource to accounts that would like to sign up and moves such a resource from accounts that would like to de-register.

To specify such modifications of the global state, we introduce the `moves_to` and `moves_from` keywords that indicate which resource type is moved to which account. All moves made by a procedure need to be specified. In Section 3.1.3 we will discuss why these specifications are required to have complete framing information. The Move Prover's specification language does not have equivalent keywords. Figure 3.7 shows how these keywords are used in the specification block (lines 13 and 25). Once again, we use the `global` keyword to refer to resources stored in accounts.

Move conditions can also be conditional. Consider, for example, a non-aborting version of `sign_up` that does nothing if the sender has already signed up, as shown in Figure 3.8. Here we use an `if` to specify under what conditions the move occurs. These conditions need to be exact, in that moves occur if and only if they are annotated in the specification.

```
module Petition {
    use 0x1::Signer;

    resource struct SignedUp {}

    public fun sign_up(transaction_sender: &signer) {
        move_to(transaction_sender, SignedUp {});
    }

    spec fun sign_up {
        aborts_if exists<SignedUp>(
            Signer::spec_address_of(transaction_sender));
        moves_to global<SignedUp>(
            Signer::spec_address_of(transaction_sender));
    }

    public fun deregister(transaction_sender: &signer) {
        let address = Signer::address_of(transaction_sender);
        let SignedUp {} = move_from<SignedUp>(address);
    }

    spec fun deregister {
        aborts_if !exists<SignedUp>(
            Signer::spec_address_of(transaction_sender));
        moves_from global<SignedUp>(
            Signer::spec_address_of(transaction_sender));
    }
}
```

Figure 3.7: Specification of resource moves

```
public fun sign_up(transaction_sender: &signer) {
    let sender_address = Signer::address_of(transaction_sender);
    if (!exists<SignedUp>(sender_address)) {
        move_to(transaction_sender, SignedUp {});
    }
}

spec fun sign_up {
    moves_to if(exists<SignedUp>(
            Signer::spec_address_of(transaction_sender)
        ))
        global<SignedUp>(
            Signer::spec_address_of(transaction_sender)
        );
}
```

Figure 3.8: Specification of a conditional move

```
1  module M {
2      resource struct T {
3          x: u64,
4          y: u64
5      }
6
7      public fun modify_x(t_ref: &mut T) {
8          t_ref.x = 42;
9      }
10
11     spec fun modify_x {
12         ensures t_ref.x == 42;
13         ensures t_ref.y == old(t_ref.y);
14     }
15 }
```

Figure 3.9: Framing of `modify_x`

3.1.3 Frame

As mentioned in the introduction, there are, in principle, two ways of handling the verification of functions that call other functions. Either we inline the functions called as part of our verification process and reason about the exact changes made line by line, the complexity of which may increase very quickly, or we use a function's specifications, making the verification process modular. While the first approach is used by the Move Prover for procedures that modify values used by the caller procedure, we use the second approach for our verifier.

Using a function's specifications, however, raises the question of what happens when a function specification is incomplete, in that a change made to the local or global state has not been specified. This might happen either because the developer forgot about the change or because they did not think it was relevant when they developed the function. If a caller procedure assumed that only the changes specified have been made, this could lead to unsoundness. Therefore, additional information is required. We will introduce the `modifies` keyword that specifies which global resource fields may be modified by a procedure. This way, the caller will not make any assumptions about the value of a modified location unless there is an explicit postcondition for it. A procedure has to specify all the *global* locations that it potentially modifies by adding a `modifies` specification for each one. For *local* changes, no additional specification is required since Move's type system indicates which variables are mutable. We will first consider examples of modifications of local variables before showing an example of how the `modifies` keyword is used to indicate modifications of global resources.

```
1  resource struct Player {
2      opponent: address,
3      wins: u64
4  }
5
6  fun lost(addr: address) acquires Player {
7      let opponent_addr: address = borrow_global<Player>(addr).opponent;
8      let opponent_ref = borrow_global_mut<Player>(opponent_addr);
9      opponent_ref.wins = opponent_ref.wins + 1;
10 }
```

Figure 3.10: Indirect account resource modification

Mutable References

Let us once again consider the example in Figure 3.9. Move's type system allows the values behind `t_ref` to be changed, as it is a mutable reference, but it also guarantees that a procedure with `modify_x`'s signature does not modify anything else. This guarantee comes from the fact that the `borrow_global_mut` and `move_from` operations cannot be executed without specifying the respective resource type in the procedure's `acquires` list. We also cannot move resources to an account without a `&signer` argument passed into the procedure. Thus, the global state may not be modified by the procedure with the exception of modifications of `t_ref`.

In the example in Figure 3.9, only the `x` field of the resource is modified. A careless developer may forget to specify that the `y` field remains unchanged. When verifying a procedure that calls `modify_x` we should, however, not assume that this is the case. If `t_ref.y` was in fact modified and the developer forgot to specify that change, this assumption could lead to the unsoundness of our verification.

We can conclude that when dealing with a mutable resource argument, the caller should assume that any of its fields might be changed.

Global Borrows

The second way of modifying the program state is by modifying an account resource, which involves a `borrow_global_mut` operation. While the `acquires` list tells us which resources may potentially be modified, this information is not particularly fine-grained and we cannot derive from the function signature which exact resources will be modified. That is because the account addresses passed into the `borrow_global_mut` operation may come from three sources:

- `&signer` or `address` arguments
- constant addresses in code
- resource `address` fields

```
1  spec fun lost {
2      aborts_if !exists<Player>(addr);
3      aborts_if !exists<Player>(global<Player>(addr).opponent);
4      aborts_if global<Player>(global<Player>(addr).opponent).wins
5          == max_u64();
6      modifies global<Player>(global<Player>(addr).opponent).wins;
7      ensures global<Player>(global<Player>(addr).opponent).wins ==
8          old(global<Player>(global<Player>(addr).opponent).wins) + 1;
9  }
```

Figure 3.11: Specification of the `lost` procedure in Figure 3.10

While the first two sources can be derived from a procedure's signature and code in a relatively straightforward way, the last source is difficult to reason about. Deriving exact expressions for the addresses borrowed from in a language with unbounded loops is potentially intractable and offers no advantage over inlining the procedure.

Consider, for example, the resource and procedure in Figure 3.10. Suppose we develop a game with a `Player` resource that keeps track of the current opponent and the number of previous wins. After a player has `lost`, we need to update their wins while the player's resource remains unchanged. At first sight, however, the signature `fun lost(addr: address)acquires Player` might lead one to believe that it is the `Player` resource at address `addr` that might be modified, which would thus define our frame.

Since we cannot make assumptions like this, the only safe over-approximation of the resources modified is that all resources of types listed in the `acquires` list are modified *at all accounts where such a resource exists*. This over-approximation would render writing specifications impossible since the developer would have to manually specify everything that has *not* changed. While we could still reason about procedures that do not borrow any global resources, the set of procedures that could be verified would be very limited.

To remedy this lack of framing information, we require the developer to specify every global resource field that is modified with a `modifies` specification. Every global resource field that may potentially be modified must be indicated by a `modifies global<Type>(address).field_name` annotation. The verification of a procedure will fail if a global resource field is modified that has not been mentioned. Instead of using multiple `modifies` clauses for the fields of one resource, the developer may also indicate that all of them may change by writing `modifies global<Type>(address)`. The complete specification of the `lost` procedure required for our verifier is shown in Figure 3.11 with the `modifies` specification on line 6. The Move Prover does not have an equivalent to the `modifies` keyword.

```
module Bank {
    use 0x1234::Currency;

    resource struct BankBalance {
        amount: u64
    }

    resource struct Reserve {
        coin: Currency::Coin
    }

    // procedures
}
```

Figure 3.12: Resources defined by a bank module

Moves

The other part of a procedure's frame we have to consider is whether a resource has been moved to or from an account. This information is required for the verifier to decide whether operations such as `borrow_global` may fail after a call to a procedure that moves resources from or to an account has been made. For this reason, we require procedure annotations that determine under what conditions resources are moved to/from accounts. We have already seen the `moves_to` and `moves_from` keywords that may be part of a postcondition (Section 3.1.2). We use these specifications to reason about the new set of accounts that hold resources of any given type to complete the frame.

3.2 Global Module Properties

In the previous section, we saw how we can specify Hoare triples for module procedures and which additional specifications are required for complete framing information. While these suffice to reason about the changes made to the global state by a procedure or a transaction, developers may not always want to reason about the exact changes made but may be more interested in meaningful properties of the state.

3.2.1 Module Invariants

As we will see in this section, Move's type system lends itself to invariants on a module level. Such invariants can then express properties about the global behaviour of a module. The approach we have taken to reasoning about global module behaviour is similar in principle but different in practice to the approach taken by the Move Prover. This section will explain and contrast the two.

As a motivating example, consider Figure 3.12, which shows the resources

defined by a bank module. The bank is supposed to work as follows: when a Libra account deposits money (`Coin`) into their bank account, their `BankBalance` is increased by the respective amount and the `Coin` is stored as part of a reserve `Coin`. When a Libra account withdraws `Coin`, this is taken from the reserve `Coin`, which stored at a fixed address. In order to be confident that `Coin` is never lost, a suitable invariant to verify is that the sum of bank balances across accounts equals the amount of `Coin` stored in the reserve.[3]

Let us recall that resources act as linear types outside the module that defines them. This renders them immutable outside their module, meaning that we only need to reason about resource modifications[4] inside the resource module. In particular, an expression referencing resources defined in one module is invariant if it holds after the execution of every public procedure in that module. The syntax for defining a module invariant is as follows:

```
1 spec module {
2     invariant /* some expression */;
3 }
```

In the next section, we will describe the body of the invariant.

3.2.2 Global Resource Properties

As we will see in Chapter 4, there is a natural way to reason about the set of accounts holding a particular resource.

Let us now think about how to encode the invariant that the total bank balances are equal to the amount of `Coin` in reserve in the `Bank` module shown in Figure 3.12.

We already know how to refer to the field of a global variable, so in order to refer to the `amount` field in the `Reserve` resource, we write `global<Reserve>(0x1234).amount`. Reading this value only makes sense if the resource exists so we require `exists<Reserve>(0x1234)`. Next we would like to sum over the `amount` fields in all `BankBalance` resources. For this we introduce a `sum` function and write `sum<BankBalance>().amount`. Lastly, we need to ensure that the invariant holds in the initial state before any public procedure is called, i.e. before any resources have been created. This is achieved by the premise that there is at least one `BankBalance`. To count the number of resource holders of a type, we introduce the `count` function. We express our condition by writing `count<BankBalance>()> 0`. Lastly, we should specify

[3] Such a bank might not be very useful, as the bank cannot use or loan any of the coin in its reserve. We might instead establish an invariant that account holders can always withdraw some percentage of their bank balance.

[4] including moving from/to an account

```
1 spec module {
2     invariant count<BankBalance>() > 0 ==>
3         exists<Reserve>(0x1234) &&
4         sum(global<BankBalance>.amount) == global<Reserve>(0x1234).
              amount;
5     invariant !exists<Reserve>(0x1234) ==> sum<BankBalance>() == 0;
6 }
```

Figure 3.13: Invariant on the Bank module

that if the reserve has not be initialised, we don't expect there to be any bank balances. One way of doing this is to specify `!exists<Reserve>(0x1234)==> sum<BankBalance>()== 0`. The complete invariant is shown in Figure 3.13.

3.2.3 The Move Prover Approach

The Move Prover does not support reasoning about sets of resource holders; instead, it supports the creation of *ghost variables*. Ghost variables are variables have no impact on the execution of Move code but they are being kept track of as symbolic values, which are updated when a resource is packed or unpacked. This means that they do not keep track of global resources as such but of all resources including local ones.[5] In Figure 3.14, we see an example of how the Move Prover uses this approach to establish an invariant. We first create global ghost variables in lines 2 and 3. Then, we specify that for `BankBalance` resources, a packing operation should lead to an increment of the `bank_balance_count` and an increase of the `total_bank_balance` (lines 11-12 and 16-17). Similarly, the ghost variables are updated for unpacking operations (lines 13-14 and 18-19). In the invariant (lines 5-7), we can reference the ghost variables. Ghost variables are not part of our specification language.

3.3 Example: Referendum Module

As a summary of the specifications we have dealt with so far and to see what a verification might look like in practice, let us take a look at a module implementing a referendum (Figure 3.15).

Every account may either vote in favour or against using the `vote` procedure. This procedure sends a `VoteToken` resource to the transaction signer, preventing them from voting again[6]. The number of votes in favour and against are counted in a `Counter` resource at a special address (`0x12345678`). Before

[5] Resources need to be destroyed or moved to an account before the end of a transaction, however, so this distinction is not that important.

[6] Attempting to send another `VoteToken` resource to the same account would abort.

```
spec module {
    global bank_balance_count: u64;
    global total_bank_balance_sum: u64;

    invariant bank_balance_count > 0 ==>
        exists<Reserve>(0x1234) &&
        total_bank_balance_sum == global<Reserve>(0x1234).amount;
}

spec struct BankBalance {
    invariant pack bank_balance_count =
        bank_balance_count + 1;
    invariant unpack bank_balance_count =
        bank_balance_count - 1;

    invariant pack total_bank_balance_sum =
        total_bank_balance_sum + amount;
    invariant unpack total_bank_balance_sum =
        total_bank_balance_sum - amount;
}
```

Figure 3.14: Invariant on the Bank module in the Move Prover style

```
module Referendum {
    use 0x1::Signer;

    spec module {
        invariant forall a: address : a != 0x12345678 ==>
            !exists<Counter>(a);
        invariant count(global<VoteToken>) > 0 ==>
            exists<Counter>(0x12345678);
        invariant count(global<VoteToken>) > 0 ==>
            count(global<VoteToken>) ==
                global<Counter>(0x12345678).in_favour +
                global<Counter>(0x12345678).against;
    }

    resource struct Counter {
        in_favour: u64,
        against: u64
    }

    resource struct VoteToken {}

    public fun init(account: &signer) { /* ... */ }

    public fun vote(account: &signer, in_favour: bool) acquires Counter
        {
        /* ... */
    }
}
```

Figure 3.15: Referendum module

```
module Ticket {
    use 0x1234::Currency;

    resource struct T {}

    public fun buy(transaction_sender: &signer) {
        let ticket_cost = Currency::withdraw(transaction_sender, 50);
        Currency::deposit(0x5678, ticket_cost);
        move_to(transaction_sender, T {});
    }
}
```

Figure 3.16: `Ticket` module using the `Currency` module

voting begins, the account with this address will need to call the `init` procedure, which aborts if the account address is unexpected and initialises and moves to the account a `Counter` resource otherwise.

There are three global properties we would like to specify:

- The `0x12345678` address should be special in that it contains the only vote `Counter`. For this property we introduce a `forall` specification that allows us to reason over the space of all account addresses.
- It is important that accounts are not allowed to vote before the `Counter` is initialised.
- To make sure that double voting is prevented, we can compare the number of `VoteToken` resources to the votes counted. `VoteToken` resources are necessarily unique at each address.

In this way we can use the linearity of resources to establish strong guarantees about the behaviour of Move programs which exploit the semantics of resource types.

3.4 Encapsulation and Reasoning Across Modules

Thus far, our specifications have been limited to individual modules. Move modules can and necessarily will, however, interact with each other. This raises the question of how much knowledge one module should have about the inner workings of another.

3.4.1 Information Hiding

Ideally, modules should not be required to know any information about how other modules work. The specifications we write for our procedures still need to be complete, though. Consider the `Ticket` module in Figure 3.16. A transaction sender may obtain a `Ticket::T` resource by calling the `buy` procedure which automatically withdraws 50 `Coin` from the sender and

```
module Currency {
    resource struct Coin {
        amount: u64
    }

    spec module {
        define value(coin: Coin): u64 {
            coin.amount
        }
    }

    // ...
}
```

Figure 3.17: Coin value specification function

deposits them at a special address (`0x5678`).

Abort conditions

The calls to the `Currency` module may fail if either the sender does not have any/sufficient `Coin` or if the special address has not been initialised properly. Like "local" abort conditions these abort conditions need to be specified. Specifying a condition like `aborts_if global<Currency::Coin>(Signer::spec_address_of(transaction_sender)).amount < 50` would, however, require the `Ticket` module to know implementation details of the `Currency` module, in this case the fields of the `Coin` resource.

Specification Functions

To remedy this lack of modularity, we prevent modules from reasoning about fields of resources they do not define and instead require modules to provide pure (non-mutating) *specification functions* that may return the fields of resources or other values. In this case, the `Currency` module would have to provide a `value` specification function returning the amount in a `Coin` resource as shown in Figure 3.17. The abort condition for insufficient funds in the `Ticket` module would then be `aborts_if Currency::value(global<Currency::Coin>(Signer::spec_address_of(transaction_sender)))< 50`. Such specifications functions should also be used in postconditions that reason about changes made by other modules.

Modifies Specifications

A similar problem arises when specifying resource fields that may have been modified. Therefore, we prevent developers from specifying fields and limit them to specifying which resources are modified. In this case, the `buy` procedure would have to specify it `modifies global<Currency::Coin>(Signer::spec_address_of(transaction_sender))` and `modifies global<Currency::Coin`

`>(0x5678))`. In the case of a `Coin` resource, this changes very little as it only has a single field. For resources that have more fields, a module caller may have to specify, using specification functions, what remains unchanged if this information is important.

There is necessarily a trade-off between verbosity of specifications and modularity of specifications here. Since our verification approach focusses on modularity, we believe it is important to also capture this in the specifications.

3.5 Transaction Script Specification

Now that we have concluded our analysis of how to specify the behaviour of modules and proposed a specification language to capture the behaviour, we need to think about how we would like to specify transaction scripts.

This is quite straightforward. As transaction scripts only have a single `main` procedure, we can treat a transaction script as a module with a single public procedure that makes calls to other procedures. The specifications we can then use to reason about changes made to the global state are those that we have used for the specification of normal module procedures.

It may be more common for the developer of a transaction script to verify that the transaction does not abort than for the developer of module, as a transaction sender has to pay the execution of every operation with gas. For this purpose, the developer of a transaction script may specify the condition `aborts_if false` to capture the behaviour of a non-aborting transaction.

Chapter 4

Encoding to Viper

In the previous chapter, we saw how the behaviour of Move modules and transactions can be captured by specifications. Now, we need to verify that the behaviour of an annotated Move program follows its specification. We do this by translating Move programs and their annotations into the Viper intermediate verification language. Once a Viper program is specified, a verification backend either confirms correctness or indicates where the program might fail.

In this chapter, we will first introduce a subset of the features of the Viper language to gain an understanding of how we can use Viper to verify programs. In subsequent sections, we will then describe in detail how we translate Move programs and their specifications to Viper. Initially we will show a simplified encoding, before adding the necessary details towards the end of the chapter.

4.1 Background on Viper

4.1.1 Structure of a Viper Program

Methods

A Viper program consist of *methods*[1] which

- are annotated with preconditions (using a `requires` clause) and postconditions (using an `ensures` clause),
- have parameters,
- return values,

[1] To avoid any confusion between Viper and Move code, we refer to *methods* in Viper and *procedures* in Move.

```
1  field value: Int
2
3  method set(x: Ref)
4      requires acc(x.value, write)
5      ensures acc(x.value, write)
6      ensures x.value == 42
7  {
8      x.value := 42
9  }
```

Figure 4.1: Simple Viper program modifying a heap location

- have a body of statements.

In order for Viper for verify a program, all methods need to match their specifications.

When Viper verifies an individual method, it assumes that the preconditions hold, i.e. it assumes a call to the method will only be made from states where the preconditions hold true, and then makes sure that, no matter the exact state, after the execution of the body the postconditions hold true.

The verification of methods is modular, in that when one method calls another, Viper only checks that the called method's preconditions are satisfied and then assumes that called method's postconditions hold true.

Fields and the Global State

Since methods are verified in isolation, we only need to consider the state of a method when describing the global state. The state of a method consists of:

- the variables in scope and their values,
- the values of field locations referenced by the variables
- access permissions to those field locations.

We will discuss access permissions in the next section. Let us now consider our first Viper program in Figure 4.1. The `set` method takes a single argument `x` of type `Ref`. Viper references can be thought of as a model for some object – they are commonly used to model a heap location. For this purpose, references can access fields as shown in line 8 (ignore lines 4-5 for now). In this case, we want to set the `Ref`'s `value` to 42, which we also annotate with the postcondition in line 6. We have thus seen the use of a variable and of a field location. In addition to parameter variables, methods may also introduce variables – using the `var` keyword – and return variables, as we will see later on.

```
1  method set(x: Ref)
2      requires acc(x.value, 9/10)
3  {
4      x.value := 42
5  }
```

Figure 4.2: Failing Viper method due to missing permission

```
1  method set(x: Ref)
2      requires acc(x.value, 9/10)
3      ensures acc(x.value, write)
4      ensures x.value == 42
5  {
6      inhale acc(x.value, 1/10)
7      x.value := 42
8  }
```

Figure 4.3: Inhaling the missing access permission

4.1.2 Access Permissions

The reader may have noticed the `acc(x.value, write)` precondition and postcondition of the `set` method. Access permissions specify which locations of the heap may be accessed at any point in the program. In this case, the `acc(x.value, write)` precondition specifies that the method has *write* access to the `value` field of `x`, a requirement to execute the assignment on line 8. While the assignment would fail if line 6 we missing, line 7 will only become important when the method is called by other methods. Since there is no reason why `set` should "consume" access permissions, other methods will rely on access permissions remaining unchanged when calling `set`. If, however, we wanted to encode that `set` can only be called once, e.g. for a static variable, we could drop the access permission postcondition.

Permission Fractions

Viper does not only support field `write permissions`. In fact, an access permission is a fractional value between zero and one (inclusive). A permission of zero indicates that no read or write access is given while a permission between zero and one (exclusive) gives read access to a location. Lastly, a permission of one represents write access. The program shown in Figure 4.2 would thus fail because a one tenth permission is missing for write access to `x.value`. The value at location `x.value` may only be read by the method.

Access permissions do not necessarily have to come from preconditions. Instead they may be *inhaled* or *exhaled* at any point in a method. Viper then adds (in the case of `inhale`) or subtracts (in the case of `exhale`) the field access permission to previously established access permissions to the field. Figure 4.3 shows how inhaling permissions to `x.value` allows us to modify it.

```
1  field x: Int
2
3  method unsound(r: Ref)
4      requires acc(r.x, write) // permission to r.x: 1
5  {
6      inhale acc(r.x, 1/3) // permission to r.x: 4/3
7      assert false // we can now prove anything
8  }
```

Figure 4.4: An unsound Viper program

```
1  method permssions_and_aliasing(x: Ref, y: Ref)
2      requires acc(x.value, 2/3)
3      requires acc(y.value, 2/3)
4  {
5      assert x != y
6  }
```

Figure 4.5: Permissions and aliasing

We will now see the two main purposes that permissions serve. Both are based on the requirement that access permissions are always represented by a value between zero and one. In fact, nothing stops a careless programmer from inhaling a permission value of greater than zero. Since Viper's reasoning is based on permission values not exceeding one, this would lead to unsound reasoning as shown in Figure 4.4.

Aliasing

The first main purpose that permissions serve is the ability to encode the aliasing of references in a natural way. Consider for this the example in Figure 4.5. The method has access permission values of `2/3` for locations `x.value` and `y.value`. Now suppose that `x` and `y` were aliasing references. In that case, Viper would sum up the permissions to find a permission value of `4/3` to `x.value`. This is impossible, so `x` and `y` cannot alias and we can make the *assertion* in line 5. If, instead, we had required `acc(x.value, 1/2)` and `acc(y.value, 1/2)`, the assertion would have failed as a combined (due to aliasing) permission value of one is possible.

Viper automates this kind of reasoning, providing a natural way to handle matters of aliasing, which is typically a challenge in formal verification.

Establishing a Frame

Just as importantly, access permissions enable us to implicitly establish a method's frame. Consider the call made in Figure 4.6. While the `caller` has no direct information about what the `callee` does with `x`, the difference in access permissions to `x.value` between the `caller` and the `callee` indicates

```
1  method caller(x: Ref)
2      requires acc(x.value, 1/3)
3  {
4      // 1/3 permission to x.value
5      callee(x) // 1/3 - 1/4 permission to x.value
6      // 1/3 permission to x.value
7  }
8
9  method callee(x: Ref)
10     requires acc(x.value, 1/4)
11     ensures acc(x.value, 1/4)
```

Figure 4.6: Permissions and framing

- `Int`
- `Bool`
- `Ref`
- `Set`
- `Perm`

Figure 4.7: Viper's Types

```
1  function value_greater_than_5(x: Ref): bool
2      requires acc(x.value, 1/3)
3      ensures result == x.value > 5
4  {
5      x.value > 5
6  }
```

Figure 4.8: Viper function

to the `caller` that `x.value` cannot change. The `callee` method may very well inhale additional access permission to `x.value` but these can never amount to write permissions since the `caller` retains 1/3 - 1/4 access.

In this way, access permissions implicitly set up the frame of methods called.

4.1.3 Types

We have already seen Viper's `Int` and `Ref` types. We will also use `Bool` to represent booleans and use Viper's generic `Set[T]` type that automatically encodes set properties (Figure 4.7). Lastly, Viper has a `Perm` type which represents a permission amount. In the next section, we will see an example of how this type is used.

4.1.4 Functions

In addition to methods, Viper also supports *functions*. Like methods, functions have preconditions and postconditions. However, they cannot modify the variable values or access permissions; they are pure. Figure 4.8 shows an example of how we define a function in Viper. Here, the function returns a boolean value indicating whether the value field of the `Ref` passed into the function is greater than five. Note that functions do not require a body and in this case the postcondition on its own fully captures the desired behaviour. Since Viper functions cannot change the program state, we can freely use them within methods and also method preconditions and postconditions. Note also that we do not need to (and cannot) specify permission postconditions as functions cannot consume access permissions, so all permission required will automatically be returned.

One important function that we will use in our encoding is the `read$()` function shown below. It returns an access permission amount. We do not specify an exact fraction but, instead, describe a symbolic value. Its value will be greater than zero but less than a write permission. The way Viper interprets the result of the `read$()` function is that any multiple of the result will still be smaller than a full write permission.

```
1 function read$(): Perm
2   ensures none < result
3   ensures result < write
```

4.1.5 Quantifiers and Quantified Permissions

So far, the Viper expressions we have seen have only involved individual `Ref`s and fields. Shortly we will see that this is not expressive enough for the state of a Move program, as we deal with an unbounded number of accounts and resources. For those purposes, Viper supports *quantifiers*, which are of the following form.

```
1 forall variables :: {trigger} body
```

If, for example, we have a method that accepts a set of integers `s` which all need to be positive, we would specify this as

```
1 requires forall i: Int :: { i in s } i in s ==> i > 0
```

The *trigger* expression helps Viper establish under which circumstances we might want the body expression to be applied to an individual variable.[2] We call this an instantiation of the quantifier. A trigger may contain multiple, comma-separated expressions, which all need to co-occur in order to trigger,

[2] If no trigger is specified, Viper will try to figure one out, but there is no guarantee that this trigger is correct or efficient.

A Move Resource

```
resource struct T {
    int_field: u64,
    bool_field: bool
}
```

Encoding to Viper

```
field T$int_field: Int
field T$bool_field: Bool
```

Figure 4.9: Encoding of a Resource Struct

and a quantifier may also contain multiple triggers, each within curly brackets, where the occurrence of the expressions in at least one trigger satisfies for Viper to instantiate the quantifier.

When reasoning about a set of references, we will usually need to specify what permissions we have to the reference's fields. Suppose that we require write access to all `value` fields of the references in a set `s`. The *quantified permission* precondition in this case would be:

```
requires forall r: Ref :: { r in s } r in s ==> acc(r.value, write)
```

4.2 Resources and Resource Semantics

4.2.1 Resource Fields

We will now begin our encoding of Move programs and their specifications to Viper. We start by encoding resource types defined in a module.

We recall that Move resources consist of fields, which may be referenced from a resource. The overlapping naming of *field* in Move and Viper might suggest to the reader that there is a natural encoding and, indeed, we encode Move resource fields as Viper fields. This implies that resources should be encoded as `Ref`s, as only `Ref`s may access fields. To ensure that field names are unique, we prefix them with the resource's name separated by the `$` symbol.[3] This can be seen in Figure 4.9.

4.2.2 Types

The encoding of the `u64` and `bool` types is straightforward, as seen in Figure 4.9.[4] We have also established that we should encode resources as `Ref` types in order for them to be able to access fields.

[3] As we will be generating one Viper file containing all modules' contents, we should also include the module's name. For readability we will omit this here.

[4] Recall from Chapter 2 that Move automatically checks to integers bounds and aborts transactions in the case of under/overflow. We will encode these checks at the point of reading and writing `u64` variables.

Moving to a signer

```
1 move_to<T>(sender, t);
```

Encoding to Viper

```
1 if (sender.val_address in
      $T_holders) {
2     // abort
3 }
4 // add the address to the
      holders
5 $T_holders := $T_holders union
      Set(sender.val_address)
```

Figure 4.10: Encoding of a Resource Moving

One type that that we have not encoded yet is the `address` type. We saw in Chapters 2 and 3 that addresses may be used to acquire, modify and move resources, and that they may be used to establish the identity of the transaction sender. Hence, our encoding of addresses needs to enable a mapping to resources and support the comparison operator. Indeed, we can use Viper's `Ref` type for this.

The last[5] type we will encode is Move's `&signer` type. We know that signers have an address, so we will encode them as a `Ref` and access their `val_address` field, a global field that will be in all programs, for the underlying address.

4.2.3 Resource Operations

Global resources may be moved from and to senders, acquired and modified. Our Viper encoding must contain a representation of the state of global resources in order for us to be able to encode changes to the global state.

Moving

Consider, for example, a method that moves a resource to the transaction sender (Figure 4.10). This operation may fail if a resource of the same type already lives at the account. In order to encode this operation, we, therefore, need a representation of the set of accounts (addresses) that hold a particular resource. As we will see, this *holders* variable is sufficient to encode other resource operations as well.

Acquires

When acquiring a resource or moving a resource from an account into a procedure, the transactions aborts when the global resource does *not* exist. Therefore, we encode `borrow_global` operations in a similar way. Remember

[5]We will not consider the encoding of Move vectors. Since they are injected into Move

Listing 4.1: Acquiring a Resource

```
1 let t_ref: &mut T =
      borrow_global_mut<T>(addr);
2 t_ref.value = 42;
```

Listing 4.2: Encoding to Viper

```
1 if (!(addr in $T_holders)) {
2     // abort
3 }
4 $var_t_ref := addr
5 $var_t_ref.T$value := 42
```

Figure 4.11: Encoding of a Resource Acquisition and Modification

that we treat resources simply as `Ref`s and that when we acquire a resource at an address, we can access the `Ref`'s fields. An example of this is shown in Figure 4.11.

Global Resource Access Permissions

In order to to be able to modify `$var_t_ref.T$value`, the method needs to have write access to this field. As established in Section 3.1.3, the developer has to specify which global resource fields may be modified and we will derive write access permissions from those annotations. We will look at the exact encoding of write permissions in Section 4.3.4. Developers do not, however, need to specify which global resource fields may be *read*. We know that if a resource exists at an address and, hence, if a `borrow_global` operation does not fail, we should automatically be granted read access to the resource's fields.

To achieve this automatically in Viper, we can specify the following property as a precondition: if an address is in the holders set of some resource type, then we are granted read permissions to the resource's fields. The encoding is shown below. When we introduce references to global resource fields later on, we will require more complex preconditions.

Encoding of Global Resource Access

```
1 requires (forall $r: Ref :: { $r in $T_holders } $r in $T_holders
2     ==> acc($r.T$field1, read$()))
3 requires (forall $r: Ref :: { $r in $T_holders } $r in $T_holders
4     ==> acc($r.T$field2, read$()))
5 // ...
```

Note that we have not specified any particular fractional permission value but, instead, expressed this as the abstract `read$()` permission. Using any one particular value would cause problems for our encoding later on, as we shall see in Section 4.3.5. For now let us just treat it as an arbitrary value between zero and one (exclusive).

as a native type with native operations, this requires a custom encoding using Viper *domains*. For this thesis, however, vectors are not a focus.

```
module Game {
    use 0x1::Signer;

    resource struct Player {
        strength: u64,
        alive: bool
    }

    // donates the donor's strength to the beneficiary
    public fun donate(donor: &signer, beneficiary: address) acquires
        Player
    {
        // read donor strength
        let donor_ref: &Player = borrow_global<Player>(Signer::
            address_of(donor));
        let donor_strength = get_strength(donor_ref);

        // read beneficiary strength
        let beneficiary_ref: &Player = borrow_global<Player>(beneficiary
            );
        let beneficiary_strength = get_strength(beneficiary_ref);

        // calculate new strength
        let new_strength = donor_strength + beneficiary_strength;

        // set beneficiary strength to new strength
        let mut_beneficiary_ref: &mut Player = borrow_global_mut<Player
            >(beneficiary);
        mut_beneficiary_ref.strength = new_strength;

        // set donor strength to zero
        let mut_donor_ref: &mut Player = borrow_global_mut<Player>(
            Signer::address_of(donor));
        mut_donor_ref.strength = 0;
    }

    spec fun donate {
        modifies global<Player>(Signer::spec_address_of(donor)).strength
            ;
        modifies global<Player>(beneficiary);
    }

    fun get_strength(player: &Player): u64 {
        player.strength
    }
}
```

Figure 4.12: Game Module

4.3 Procedures, Aliasing and Frame

In the previous section, we established how we encode resource properties and operations into Viper. We will now use this for our encoding of Move procedures to Viper methods. Apart from the encoding of aborts conditions and postconditions, which we will cover in the next section, we have to encode:

- parameters,

donate signature

```
1 public fun donate(
2     donor: &signer,
3     beneficiary: address)
4     acquires Player
```

Encoding to Viper

```
1 method $donate(
2     $donor: Ref,
3     $beneficiary: Ref,
4     $Player_holders: Set[Ref])
5     requires acc($donor.
          val_address, read$())
6     ensures acc($donor.
          val_address, read$())
```

Figure 4.13: Encoding of `donate` parameters

- global resource access permissions,
- write permissions,
- return values,
- the procedure body.

Throughout the following subsections we will encode the Move module shown in Figure 4.12. The module represents some game where `Players` have `strength` which they may `donate` to another `Player`. This example is purposefully tricky: if a player tries to donate strength to their own address, the procedure may not behave as expected. This will be a good test to see how our encoding deals with aliasing and later on what the specification of the procedure should look like.

4.3.1 Parameters

Encoding of Parameters in Move

Let us start by formulating the parameters of the `$donate` Viper method (shown in Figure 4.13). In the first step we add all Move procedure parameters to the method. The `&signer` type is encoded as a `Ref` and to ensure that we have access to the signer's address, we add an `acc($donor.val_address, read$())` precondition. Since we do not "consume" this access permission, we also add it as a postcondition. The `address` type is simply encoded as a `Ref`.

Global Resource Holder Parameters

In the next step we need to establish which global resource types we want to use in this method. In this case we only use `Player` resources, so we add a `$Player_holders` parameter. In general, we need holder parameters for resource types

- acquired (borrowed globally)

get_strengh signature

```
1 public fun get_strength(
2     player: &Player): u64
```

Encoding to Viper

```
1 method $get_strength(
2     $player: Ref)
3     requires acc($player.
          Player$strength, read$()
          )
4     requires acc($player.
          Player$alive, read$())
5     ensures acc($player.
          Player$strength, read$()
          )
6     ensures acc($player.
          Player$alive, read$())
```

Figure 4.14: Encoding of the `get_strength` parameter

- moved globally
- mentioned in the specification.

Encoding of Resource Parameters

Since none of these apply to the `get_strength` procedure, we only need to do translate the `player` parameter, which is of type `&Player`. As this reference is immutable, in Move we can only read the resource's fields. In Viper we can naturally encode this as a read permission to the resource's fields, as shown in Figure 4.14. As before, we do not consume those access permissions, so we specify them also in the postconditions.

If our method, instead, were passed a *mutable* reference to a resource or simply a resource itself, we have to encode the permission to change the resource's fields and would thus replace the `read$()` permissions by `write` permissions. As we recall from Figure 2.13, a resource or a mutable reference to a resource cannot alias with any other resource or reference to one.[6] Our encoding of field access permissions, therefore, naturally encodes this property of Move's type system. In contrast, immutable references may alias each other, which we account for with a `read$()` permission. This also shows why we use a *symbolic* read permission instead of any particular fractional value. A value of 1/2 would, for example, imply that of any three immutable references there must be two which cannot alias.

While a module cannot directly modify resources it does not define, write permissions to those types may still be required. Recall that the module may call procedures in another module, which may then modify the resource.

[6] Here, and also in general, we treat mutable references to resources and resources themselves (of the same type) identically (as `Ref`s with field access permission). This is because their semantic properties are identical.

Nested resource parameter

```
1  resource struct S {
2      field: u64
3  }
4
5  resource struct T {
6      s: S
7  }
8
9  public fun foo(t_ref: &mut T) /*
       ... */
```

Encoding to Viper

```
1  field S$s_field: Int
2  field T$s: Ref
3
4  method foo($t_ref: Ref)
5    requires acc($t_ref.T$s, write
         )
6    requires acc($t_ref.T$s.
         S$field, write)
7    ensures acc($t_ref.T$s, write)
8    ensures acc($t_ref.T$s.S$field
         , write)
```

Figure 4.15: Encoding of a nested-resource parameter

`get_strength` signature

```
1  public fun get_strength(
2      player: &Player): u64
```

Encoding to Viper

```
1  method $get_strength(/* ... */)
2      returns ($abort: Bool,
3          $RES_0: Int)
4      /* ... */
```

Figure 4.16: Viper `$get_strength` return variables

Therefore, the caller should expect resource arguments or mutable reference to resource arguments to be potentially modified.

So far we have only considered the simple case of resources that contain fields of primitive types in our encoding. Recall, however, that resources may also contain other resources. We encode access to these contained resources in the same way. An example of this is shown in Figure 4.15. This strategy of recursively encoding resource accesses would not work in the case of cyclical resource definitions; however, these are not permitted in Move.[7]

4.3.2 Variables Returned

Viper does not have an explicit return operation (like C) or an implicit return operation (like Move). Instead, methods have a list of variables that they return. These variables are assigned as part of the method body. For Move procedures that return one or more values, we need to add return variables to the Viper method using the `returns` keyword followed by a list of variables in parentheses.

We also use such a return variable, `$abort: Bool`, to encode whether the

[7] Viper provides a mechanism to handle recursively defined data structure called *predicates*.

donate signature

```
1 public fun donate(
2     donor: &signer,
3     beneficiary: address)
4     acquires Player
```

Encoding to Viper

```
1 method $donate(
2     /* ... */,
3     $Player_holders: Set[Ref])
4     returns ($abort: Bool,
5         $new_Player_holders: Set
              [Ref]))
```

Figure 4.17: Viper `$donate` return variables

```
1 method $donate(/*...*/) /* ... */
2     requires (forall $r: Ref :: ($r in $Player_holders) ==> acc($r.
          Player$strength, read$()))
3     requires (forall $r: Ref :: ($r in $Player_holders) ==> acc($r.
          Player$alive, read$()))
4     ensures (forall $r: Ref :: ($r in $new_Player_holders) ==> acc($r.
          Player$strength, read$()))
5     ensures (forall $r: Ref :: ($r in $new_Player_holders) ==> acc($r.
          Player$alive, read$()))
```

Figure 4.18: Global Resource Access Permissions

transaction aborts. For `get_strength`, the encoding is, thus, as shown in Figure 4.16. Note that we do not need to specify access permissions for simple variables returned. If we return a resource, we will need to specify field access write permissions in the same way as we did for resource parameters.

Similar to how our Viper method has additional arguments for resource holders, our methods will also return the sets of new resource holders, called `$new_T_holders` for a resource of type `T`. For `$donate`, this is shown in Figure 4.17.

4.3.3 Global Resource Access Permissions

For those resources, the Viper method requires the global resource access permissions we have established in Section 4.2.3. We then ensure in the postconditions that these permissions still hold for the new set of resource holders.[8] We can see the code for this in Figure 4.18.

4.3.4 Write Permissions

Now that the method has read access permissions to global resources as well as read/write permissions to resource arguments, we need to ensure that the method has write permissions to the global resource fields modified. As established in Section 3.1.3, our verification process requires `modifies` annotations which identify global resource fields that may be modified.

[8] Effectively, this turns the resource access property into an invariant.

Modification of multiple resources

```
resource struct T {
    val: u64
}

public fun foo(addr1: address,
        addr2: address) {
    let ref1 = borrow_global_mut
        <T>(addr1);
    ref1.val = 0;
    // first borrow expires

    let ref2 = borrow_global_mut
        <T>(addr2);
    ref2.val = 1;
    // global<T>(addr1).val
    // might equal 0 or 1.
}

spec fun foo {
    modifies global<T>(addr1).
        val;
    modifies global<T>(addr2).
        val;
    /* ... */
}
```

Encoding to Viper

```
method foo($addr1: Ref, $addr2:
    Ref, T_holders: Set[Ref])
    requires /* field access */
    requires forall $r: Ref ::
        { $r == $addr1 }
        { $r == $addr2 }
        ($r == $addr1 ||
         $r == $addr2) ==>
        acc($r.T$val, write -
            read$())
{ /* ... */ }
```

Figure 4.19: Encoding of multiple `modifies` annotations

In addition, when a procedure moves a resource to a signer, we will also require write access to the resource's fields.

Modified Fields

Remember that we can only modify resource fields where the resource exist and that for those resources that exist, we already have read access to the fields. Therefore, in order for the Viper method to have one full write permission to a certain field, an additional `write - read$()` permission is required. Hence, the simple encoding of a `modifies` annotation is shown below.

```
modifies global<T>(addr).val;
```

```
requires addr in $T_holders ==>
    acc(addr.T$val, write -
        read$())
```

This encoding implies that the `T` resource at address `addr` cannot alias any other resource we are dealing with. While this holds true for resource parameters, as ensures by Move's type system, Move makes no guarantees about

Modification of an address

```
resource struct T {
    other: address,
    val: u64
}

public fun foo(addr: address) {
    let other: address =
        borrow_global_mut
        <T>(addr).other;
    // borrow expires

    borrow_global_mut<T>(other).
        val = 42;
    // borrow expires

    borrow_global_mut<T>(addr)
        .other = 0x12345;
    // borrow_expires
}

spec fun foo {
  modifies global<T>(global<T>(
      addr).other).val;
  modifies global<T>(addr).other
      ;
  /* ... */
}
```

Encoding to Viper

```
method foo($addr: Ref)
  /* ... */
  requires acc($addr.T$other.
      T$field, write - read$())
  requires acc($addr.T$other,
      write - read$())
  ensures acc(old($addr.T$other)
      .T$field, write - read$())
  ensures acc(old($addr).T$other
      , write - read$())
{ /* ... */ }
```

Figure 4.20: Address Modification

resources acquired *sequentially*.[9] Consider the Move example in Figure 4.19. We first borrow the `T` resource at `addr1` and modify its `val` field. Then, we repeat this for the resource at address `addr2`. Move makes no guarantee as to whether `addr1` and `addr2` can alias, so we must assume that they may. Encoding the `modifies` annotations as

```
requires addr1 in $T_holders ==> acc(addr1.val, write - read$())
requires addr2 in $T_holders ==> acc(addr2.val, write - read$())
```

would therefore we incorrect since this would not allow for aliasing. Instead, when dealing with multiple `modifies` to fields of the same type, we have to use a quantified permission as shown in the Viper encoding of Figure 4.19. Of course, only fields of resources of the same type may alias, so we need such a quantified permission for every field that may be modified.

Since we want to allow future modifications of the same fields, we need to specify the same access permissions in the postconditions. Here, we need

[9] We call two global borrows of resources of the same type sequential if the first borrow expires before the second is executed. In practice, this means that the second call to `borrow_global` or `borrow_global_mut` is made after the last usage of a reference to the first re-

Modifies Annotations

```
1 spec fun donate {
2     modifies global<Player>(
          Signer::spec_address_of(
          donor)).strength;
3     modifies global<Player>(
          beneficiary);
4 }
```

Encoding to Viper

```
1  method $donate(/* ... */)
2    /* ... */
3    requires forall $r: Ref ::
4        { $r.Player$strength }
5        (($r in $Player_holders)
             &&
6        ($r == $donor.val_address
             ||
7         $r == $beneficiary))
8        ==> acc($r.Player$strength
             ,
9            write - read$())
10   ensures forall $r: Ref ::
11       { $r.Player$strength }
12       (($r in
             $new_Player_holders)
             &&
13       ($r == old($donor.
             val_address) ||
14        $r == old($beneficiary)))
15       ==> acc($r.Player$strength
             ,
16           write - read$())
17 { /* ... */ }
```

Figure 4.21: Encoding `modifies` annotations of `donate`

to careful, though. Consider the example in Figure 4.20 where we modify the `val` field of the `T` resource at address `global<T>(addr).other`. Afterwards, we modify this address. Since we have not specified access permissions for this new address stored in `global<T>(addr).other`, the postconditions must not specify access to fields of resources at this address. Therefore, we wrap these addresses in an `old` expression, as shown in the encoding in Figure 4.20, avoiding this issue. Similar to the access permission precondition, the access permission postcondition will be quantified if multiple fields of the same type are modified.

We can now translate the modifies annotations of the `donate` procedure in our `Game` example. This encoding is shown in Figure 4.21.

Moves

The other kind of write permission we need to encode is for moving a resource to an account, which we describe as writing to the address' respective fields. In this case, we do not already have read permission to the field, so we need to require a full write permission. Note that we make sure to require access permissions only if the address is not in the set of resource

source borrowed.

Simple Move to an Account

```
1  resource struct T {
2      val: u64
3  }
4
5  public fun foo(sender: &signer)
       {
6      move_to<T>(sender, T {
7          val: 42
8      });
9  }
10
11 spec fun foo {
12     /* ... */
13     moves_to global<T>(
14         Signer::spec_address_of(
               sender));
15 }
```

Encoding to Viper

```
1  method foo(/* ... */)
2      requires !(sender.
           val_address
3          in $T_holders) ==>
4          acc(sender.val_address.
               T$val, write)
```

Figure 4.22: Encoding of Move Write Permissions

Potentially Aliasing Move and Modification

```
1  public fun foo(sender: &signer)
       {
2      move_to<T>(sender, T {
3          val: 42
4      });
5
6      borrow_global_mut<T>(0x9)
7          .val = 43;
8  }
9
10 spec fun foo {
11     /* ... */
12     moves_to global<T>(
13         Signer::spec_address_of(
               sender));
14     modifies global<T>(0x9).val;
15 }
```

Encoding to Viper

```
1  method foo($sender: Ref,
       $T_holders: Set[Ref])
2      returns ($abort: Bool,
3          new_T_holders: Set[Ref])
4      /* ... */
5      requires !(sender.
           val_address
6          in $T_holders) ==>
7          acc($sender.val_address.
               T$val, write)
8      requires (address$(9)
9          in $T_holders) ==>
10         acc(address$(9).T$val,
11             write - read$())
12     /* ... */
13     ensures forall $r: Ref ::
14         { $r.T$val }
15         (($r in $new_T_holders)
               &&
16          ($r == old($sender.
                   val_address) ||
17           $r == old(address$(9))
                ))
18         ==> acc($r.T$val, write
               - read$())
```

Figure 4.23: Encoding of a Combination of `moves_to` and `modifies` Specifications

holders – otherwise we risk inhaling a permission value greater than zero, leading to unsoundness. Thus, the encoding is shown in Figure 4.22.

Unlike address modifies permissions, we can encode these permissions individually, as signers cannot alias each other. However, we need to be careful about the method's postconditions. First, note that once we move the resource to the account, we turn the "`moves_to` permission" into a "`modifies` permission". This allows us to handle modifications of the resource after the move. Of course, it does not make sense for the developer to use a `modifies` specification for a resource at an account before it has been created. By keeping write permissions to the fields we have written to through `move_to`, we allow the procedure to continue modifying the field. While Move's type system ensures that we do not run into aliasing issues when specifying write access permissions to signers' addresses in the method's precondition, it is possible for a developer to include a `modifies` annotation for the same fields modified by the `move_to` operation. In the precondition, the `addr in $T_holders` premise ensures no harm is done, but this access permission would still form part of the postcondition where we would now specify a "double" write permission to the same field. We solve this issue by placing the field access permissions for resources moved to an account into the same quantified permission as the `modifies` permissions. An example of this is shown in Figure 4.23. Note that we make use an `address$()` function, which simply returns a unique `Ref` for every `Int`. The implementation is shown below.

```
1 function address$(a: Int): Ref
2   ensures (forall x: Int, y: Int :: x != y ==>
3     address$(x) != address$(y))
```

Conditional Moves

If the `moves_to` annotation is conditional, we add the condition to method precondition and postcondition as shown in Figure 4.24.

Moving from an Account

So far we have only dealt with the moving of a resource *to* an account. The encoding of a `moves_from` annotation is the same as that of a `modifies` annotation.[10] In the postcondition, however, we do not specify access permissions to this resource. In this case, we do consume access permissions.

4.3.5 Procedure Body

We have now translated procedure parameters and specified their access permissions as well as access permissions for global resources read and modi-

[10]For a conditional `moves_from`, we add the condition in the same way we did for `moves_to`.

Conditional `moves_to`

```
/* ... */
moves_to if(b) global<T>(
    Signer::spec_address_of(
        sender));
/* ... */
```

Encoding to Viper

```
/* ... */
requires (!(sender.val_address
    in $T_holders) && b) ==>
    acc($sender.val_address.
        T$val, write)
/* ... */
ensures forall $r: Ref ::
    { $r.T$val }
    (($r in $new_T_holders) &&
    ($r == old($sender
        .val_address) && b
     || /* ... */))
        ==> acc($r.T$val,
            write - read$())
/* ... */
```

Figure 4.24: Encoding of a Conditional `moves_to` Specification

fied and access permissions for return variables. In the next step, we will encode the body of a Move procedure. In order to simplify this process, we do not translate Move code directly. Instead, we will translate stackless Move bytecode. At this point, the difference between Move code and Move bytecode is not very important for us; in fact, they look quite similar. Notably, however, if-else branching is replaced by labels and goto operations. The other important aspect is that the byteocde we will translate is *stackless*. This means that there will be no stack operations (i.e. push and pop), but instead when referring to variables, they will have a unique identification number. The stackless bytecode statements we will have to encode are shown in Figure 4.25.

Before we start with the encoding, note that we will make references to an `end` block that will be located at the end of the method, such that a `goto end` statement encodes a return from the method.

Initialisation

We begin by initialising three kinds of variables in the Viper method:

- The `$abort` variable is set to false with `$abort := false`.
- We set the new sets of resource holders equal to the old set of resource holders with `$new_T_holders := $T_holders` for all relevant resource types.
- We declare the local variables that are referenced by the stackless Move bytecode. For a Move procedure with n parameters, the variables 1-n will refer to the arguments while the following ones we need to declare with, for example, `var $var_6: Int`. The type of the local variable will

Statement	Explanation
Assign(target, source)	Assign the source variable to the target variable.
Ret(variables)	Return one or more variables.
Load(target, constant)	Set the value of the target to the constant.
Branch(condition, label1, label2)	If the condition is true, jump to label1, otherwise jump to label2.
Jump(label)	Jump to a label.
Label(label)	Define a label.
Abort	Abort the transaction.
Function(function id, args)	Call a procedure with specified arguments.
Pack(resource id, target, sources)	Pack a resource, setting the fields equal to the sources and assign the result to the target variable.
Unpack(targets, source)	Unpack a resource (stored in source), assigning the fields to target variables.
MoveTo(res id, source, address)	Move a resource (stored) in source to the account with the given address.
MoveFrom(res id, target, address)	Move a resource from an account into the current procedure, storing it in target.
GetGlobal(res id, addr, target)	Borrow a resource from an account immutably, storing it in target.
BorrowGlobal(res id, addr, target)	Borrow a resource from an account mutable, storing it in target.
GetField(field num, target, source)	Assign the value of a field of a local resource to target.
BorrowField(res id, field num, target, source)	Borrow a field from a local resource variable, storing it in target.
BorrowLoc(target, source)	Borrow a mutable reference to a local source variable, storing it in target.
ReadRef(target, source)	Source is a reference here, store the value it refers to in the target.
WriteRef(target, source)	Set the value referenced by target equal to the source value.
Other unary and binary operations	Not, Add, Sub, Mul, Div, Eq, Lt, Le, Gt, Ge, Or, And, ...

Figure 4.25: Table of Stackless Bytecode Statements

Assignment

```
1 let new_strength = old_strength;
```

Encoding to Viper

```
1 $var_4 := $var_3
```

Figure 4.26: Encoding of Assign

depend on its usage in the bytecode. As before, we will translate `u64` as `Int`, `bool` as `Bool` and resources as well as addresses as `Ref`.

Later on, we will see that we will also need to declare temporary variables.

Non-Reference Operations

Next, let us translate the Move bytecode instructions. For each of the following operations, an example encoding (referring in style to our Game module) is shown in the figures referenced.

Bytecode	Encoding
Assign	Figure 4.26
Ret	Figure 4.27
Load	Figure 4.28
Branch	Figure 4.29
Abort	Figure 4.30
Pack	Figure 4.31
Unpack	Figure 4.32
MoveTo	Figure 4.33
MoveFrom	Figure 4.34
GetGlobal	Figure 4.35
BorrowGlobal	Figure 4.36
GetField	Figure 4.37
Arithmetic	Figure 4.38

While the figures show the encoding from Move to Viper rather than the encoding of stackless Move bytecode to Viper, Move's bytecode is sufficiently similar to enable the representation of a direct encoding. Since the BorrowLoc, BorrowField, ReadRef, WriteRef operations require the handling of references, we will show their encoding in Section 4.3.6 alongside the introduction of references.

Procedure Calls

The next operation we will encode to Viper is a call from one procedure to another. There are three parts we need to consider:

- arguments passed,

Ret

```
1 fun /* ... */ {
2     /* ... */
3     (old_strength,
4      new_strength)
5 }
```

Encoding to Viper

```
1 $RES_0 := $var_3
2 $RES_1 := $var_4
```

Figure 4.27: Encoding of Ret

Load

```
1 let old_strength = 3;
```

Encoding to Viper

```
1 $var_3 := 3
```

Figure 4.28: Encoding of Load

Branch

```
1 if (alive) {
2     /* ... */
3 } else {
4     /* ... */
5 }
```

Encoding to Viper

```
1 if ($var_6) {
2     goto if_branch_1
3 }
4 goto else_branch_1
```

Figure 4.29: Encoding of Branch

Abort

```
1 abort 100;
```

Encoding to Viper

```
1 abort := true
2 goto end
```

Figure 4.30: Encoding of Branch

Pack

```
1 let p = Player {
2     strength: new_strength,
3     alive: alive
4 }
```

Encoding to Viper

```
1 inhale acc($var_5.
      Player$strength, write)
2 inhale acc($var_5.Player$alive,
      write)
3 $var_5.Player$strength := $var_4
4 $var_5.Player$alive := $var_2
```

Figure 4.31: Encoding of Pack

Unpack

```
let Player {
    new_strength,
    alive
} = p;
```

Encoding to Viper

```
$var_4 := $var_5.Player$strength
$var_2 := $var_5.Player$alive
```

Figure 4.32: Encoding of Unpack

MoveTo

```
move_to<Player>(sender, p);
```

Encoding to Viper

```
if ($sender.val_address in
    $new_Player_holders) {
    $abort := true
    goto end
}
$sender.val_address.
    Player$strength := $var_5.
    Player$strength;
$sender.val_address.Player$alive
     := $var_5.Player$alive;
$new_Player_holders :=
    $new_Player_holders
    union Set($sender.
        val_address)
```

Figure 4.33: Encoding of MoveTo

MoveFrom

```
let p = move_from<Player>(0x9);
```

Encoding to Viper

```
if (!(address$(9) in
    $new_Player_holders) {
    abort := true
    goto end
}
$var_5 := address$(9)
$new_Player_holders =
    $new_Player_holders
    setminus Set(address$(9))
```

Figure 4.34: Encoding of MoveFrom

GetGlobal

```
let p_ref = borrow_global<Player
    >(0x9);
```

Encoding to Viper

```
if (!(address$(9) in
    $new_Player_holders) {
    abort := true
    goto end
}
$var_5 := address$(9)
```

Figure 4.35: Encoding of GetGlobal

BorrowGlobal

```
1 let p_ref = borrow_global_mut<
      Player>(0x9);
```

Encoding to Viper

```
1 if (!(address$(9) in
      $new_Player_holders) {
2     abort := true
3     goto end
4 }
5 $var_5 := address$(9)
```

Figure 4.36: Encoding of BorrowGlobal

GetField

```
1 old_strength = p_ref.strength;
```

Encoding to Viper

```
1 $var_3 := $var_5.Player$strength
```

Figure 4.37: Encoding of BorrowGlobal

Add

```
1 new_strength = old_strength +
      10;
```

Encoding to Viper

```
1 inhale $var_3 <= max_u64$()
2 $var_4 := $var_3 + 10
3 if ($var_4 > max_u64$()) {
4     abort := true
5     goto end
6 }
```

Figure 4.38: Encoding of Add

- values returned,
- establishing the frame.

The Move bytecode will contain the variables that need to be passed to the procedure called. In addition, however, we need to pass in the resource holder sets that are required by the callee. Typically, these sets are part of the holder sets required by the caller (otherwise we cannot express any properties about those resources), however, this is not necessarily the case. For the resource holders that are part of the caller, we simply pass those to the procedure. We will not need to (and, in fact, cannot) reason about the exact changes, so we create a new variable of type `Set[Ref]` for which we inhale the global resource access permissions and pass this variable to the callee.

Since methods return new resource holder sets for the resource holder parameters they require, we will assign those return values to the caller's holder set expressions. The other return values will be assigned to local variables as described by the stackless bytecode.

Argument	Permissions held	Permissions required	Mutable
Simple variables	Not applicable	Not applicable	No
Resource variable	Write permissions to all fields	Write permissions to all fields	Yes
Mutable reference to resource variable	Write permissions to all fields	Write permissions to all fields	Yes
Immutable reference to resource variable	Read permissions to all fields	Read permissions to all fields	No
Resource holder set	Read permissions for all global resource fields that exist, write permissions for fields specified	Read permissions for all global resource fields that exist, write permissions for fields specified	Only those fields at account resources specified by `modifies`

Figure 4.39: Argument Permissions Before a Procedure Call

Now, let us consider the implicit frame established through the access permissions passed to the caller and those retained. The table in Figure 4.39 shows which permissions are held by the caller, which are required by the callee and whether the argument passed should have any mutability.

We recall from Section 4.1.2 that we can encode immutability of the fields of a `Ref` passed to a method by retaining some field access permissions. Hence, we need to retain some access permissions to

- the fields of immutable resource references passed,
- the fields of the accounts in the resource holder sets passed to the callee, but not those for which we hold write permissions.

In order to retain those permissions, we simply inhale another read permission to those fields before the call and exhale them after. When inhaling global resource field read permissions, we need to exempt the fields for which we hold write permissions. We do this by *exhaling* read permissions for those before, and then inhaling them after the call. This ensures that we never hold a permission value greater than one for any field at any point. Note that the method we call may change the set of resource holders. Of course, we must not exhale read permissions for resource fields that we did not initially inhale read permissions for. Therefore, we assign the set of resource holders to a temporary variable that we then use for the exhaling of read permissions. An example of the encoding of a procedure call is shown in Figure 4.40.

4.3.6 References

Thus far, we have restricted our handling of references to the handling of resource references, where we effectively treat resources and resource refer-

Procedure Call

```
resource struct T {
    val: u64
}

/* ... */

public fun caller(t_ref: &T) acquires T {
    /* ... */
    callee(t_ref);
    /* ... */
}

spec fun caller {
    modifies global<T>(0x9).val;
}

public fun callee(t_ref: &T) acquires T /* ... */
```

Encoding to Viper

```
method $caller($t_ref: Ref, $T_holders: Set[Ref])
    /* ... */
{
    /* ... */
    inhale acc($t_ref.T$val, read$())
    var $tmp_T_holders: Set[Ref]
    $tmp_T_holders := $new_T_holders
    exhale address$(9) in $new_T_holders ==> acc(address$(9).T$val,
        read$())
    inhale forall $r: Ref :: { $r.T$val } $r in $new_T_holders ==> acc(
        $r.T$val, read$())
    $abort, $new_T_holders := $callee($t_ref, $new_T_holders)
    if ($abort) {
        goto end
    }
    exhale forall $r: Ref :: { $r.T$val } $r in $tmp_T_holders ==> acc(
        $r.T$val, read$())
    inhale address$(9) in $tmp_T_holders ==> acc(address$(9).T$val,
        read$())
    exhale acc($t_ref.T$val, read$())
    /* ... */
}
```

Figure 4.40: Encoding of a Procedure Call

ences identically in our Viper encoding.[11] We will now see that the general encoding of references requires an update to the encoding we have thus far described.

References to Local Variables

Consider the Move procedure `caller` in Figure 4.41. When modifying the variable passed to `callee`, we need to make sure that the modification of `v` by `callee` is visible to `caller`. Therefore, we need to pass a `Ref` to `callee` in our Viper encoding, which will modify one of the `Ref`'s fields.

In general, since we need to be able to support the `&` operator for local variables, all local variables will be of type `Ref`. We will inhale access permissions to the *value field* of those local variables in order to be able to set their values. An example of the updated declaration of a local variable can be seen on lines 9 and 10 of the second part of Figure 4.41.

Variable Type	Value Field
u64	val_u64
bool	val_bool
address	val_address
signer	val_address
resource	*resource fields*

When assigning local variables to one another, our encoding does not change and we assign one `Ref` to another. However, when assigning a parameter to a local variable or passing a local variable as an argument, we need to distinguish between the `Ref` and the value field. Figure 4.42 shows an example of how a `u64` variable and a `&u64` variable are declared the same way in the encoding but used differently.

References to Resource Fields

The other kind of reference that we have not handled so far is a reference to a resource field. Similar to references to local variables, Move supports immutable and mutable references to resource fields. Mutable field accesses are represented by a `BorrowField` bytecode instruction while immutable field accesses are represented as `GetField`. As with local variables, we introduce an extra `Ref` layer that has access to a value field in order to support such references. An example is shown in Figure 4.43. In addition to these operations, field references can be read from and written to with the `ReadRef` and `WriteRef` instructions. Their encodings to Viper are also shown in Figure 4.43.

[11] The only difference is that methods require field read permissions for immutable refer-

Passing a mutable reference to an integer to a procedure

```
fun caller() {
    let v: u64 = 0;
    //     BorrowLoc
    callee(&mut v);

    if (v != 42) {
        abort 100
    };
}

fun callee(v: &mut u64) {
    *v = 42;
}

spec fun callee {
    ensures v == 42;
}
```

Encoding to Viper

```
field val_u64: Int

method $caller() returns ($abort: Bool)
{
    // $abort initialisation
    $abort := false

    // local variable initialisation
    var $v: Ref
    inhale acc($v.val_u64, write)

    $v.val_u64 := 0

    $callee($v)

    if ($v.val_u64 != 42) {
        $abort := true
        goto end
    }

    label end
}

method $callee($v: Ref)
    requires acc($v.val_u64, write)
    ensures acc($v.val_u64, write)
    ensures $v.val_u64 == 42
{
    $v.val_u64 := 42
}
```

Figure 4.41: Non-Resource Reference

Variable Assignment Procedure

```
1 fun var_ass(simple: u64,
2     ref: &u64) {
3     let simple_copy: u64 =
4         simple;
5     let ref_copy: &u64 = ref;
6 }
```

Encoding to Viper

```
1  method $var_ass($simple: Int,
2      $ref: Ref) /* ... */
3      requires acc($ref.val_u64,
4          read$())
5      /* ... */
6  {
7      // simple_copy
8      var $var_3: Ref
9      // ref_copy
10     var $var_4: Ref
11     inhale acc($var_3.val_u64,
12         write)
13     inhale acc($var_4.val_u64,
14         write)
15
16     $var_3.val_u64 := $simple
17
18     $var_4 := $ref
19 }
```

Figure 4.42: Variable Assignment of Simple Types and References

Updated Encoding of the Global Resources Access Permissions

As we have changed our encoding of resources and their fields, we need to update the field access permissions of global resources (previously shown in Figure 4.18), as shown in Figure 4.44. Note that the first precondition is unchanged. The second precondition (lines 4-6) is the access permission to the value field. In quantified expressions, Viper requires receiver expressions (`Ref` expressions whose fields are accessed, in this case `$r` and `$r.Player$strength`) to be injective with respect to the quantified variable. While this is automatically the case for the quantifier `$r`, we also require this to be the case for `$r.Player$strength`, which is why we require the third precondition (lines 7-10) that establishes that for different `Ref`s, the `Player$strength` fields are different. Since Move resources cannot hold references, this encoding is also semantically sound.

For global resources that are modified (i.e. that we have write permissions to) we will similarly update our encoding and require `write - read$()` permissions to value fields but now only `read$()` permissions to the reference field. This means that we will also need to inhale and exhale permissions for the value fields when making a procedure call. The updated procedure call encoding of Figure 4.40 is shown in Figure 4.45.

Lastly, we need to update our `move_to` encoding to re-establish field injectivity when adding an address to a set of resource holders. This is shown in

ences and field write permissions for resources and mutable references.

Passing a mutable field reference

```
resource struct T {
    val: u64
}

fun field_ref(t: T) {
    // BorrowField
    let v_ref = &mut t.val;
    inc(v_ref);
    /* consume t */
}

fun inc(v_ref: &mut u64) {
 // WriteRef        ReadRef
    *v_ref        = *v_ref + 1;
}
```

Encoding to Viper

```
// value field
field val_u64: Int
field val_bool: Int
// ...

// resource fields
field T$val: Ref

method $field_ref($t: Ref) /* ... */
    requires acc($t.T$val, write)
    requires acc($t.T$val.val_u64, write)
    /* ... */
{
    var $var_2: Ref
    inhale($var_2.val_u64, write)

    // BorrowField
    $var_2 := $t.T$val

    $inc($var_2)
}

method $inc($v_ref: Ref)
    requires acc($v_ref.val_u64, write)
    /* ... */
{
    var $var_2: Ref
    var $var_3: Ref
    inhale ($var_2.val_u64, write)
    inhale ($var_2.val_u64, write)

    // ReadRef
    $var_2 := $v_ref

    $var_3.val_u64 := $var_2.val_u64 + 1

    /* skipping overflow check */

    // WriteRef
    $v_ref.val_u64 := $var_3.val_u64
}
```

Figure 4.43: Resource Field Reference

```
method $donate(/*...*/) /* ... */
    requires (forall $r: Ref :: {$r in $Player_holders}
        ($r in $Player_holders) ==> acc($r.Player$strength, read$()))
    requires (forall $r: Ref :: {$r in $Player_holders}
        ($r in $Player_holders)
        ==> acc($r.Player$strength.val_u64, read$()))
    requires (forall $r: Ref, $s: Ref ::
        { $r in $Player_holders, $s in $Player_holders }
        ($r != $s && $r in $Player_holders && $s in $Player_holders)
        ==> $r.Player$strength != $s.Player$strength

    requires (forall $r: Ref :: {$r in $Player_holders}
        ($r in $Player_holders) ==> acc($r.Player$alive, read$()))
    requires (forall $r: Ref :: {$r in $Player_holders}
        ($r in $Player_holders)
        ==> acc($r.Player$alive.val_bool, read$()))
    requires (forall $r: Ref, $s: Ref ::
        { $r in $Player_holders, $s in $Player_holders }
        ($r != $s && $r in $Player_holders && $s in $Player_holders)
        ==> $r.Player$alive != $s.Player$alive

    ensures /* ... */
```

Figure 4.44: Updated Global Resource Access Permissions

Figure 4.46.

4.4 Specifications, Functions and Global Properties

In the previous section, we defined how we encode Move procedures to Viper. In this section, we will establish our encoding of procedure specifications, specification functions and module invariants.

4.4.1 Abort Conditions

The first kind of procedure specification that we will encode in Viper is an `aborts_if` condition. Recall that the developer has to specify all conditions under which a transaction aborts. In some cases, these conditions only make sense when considered in combination with each other. Therefore, we encode the combination of `aborts_if` conditions as sufficient and necessary for the `$abort` return variable to be true. An example is shown in Figure 4.47. Note also that abort conditions always refer to the state before the procedure call, which is why we wrap the conditions in `old(...)` expressions.

4.4.2 Postconditions

Postconditions in Move specify the state after the execution of a procedure if the procedure did not abort. Therefore, all postconditions encoded in Viper will of the form `ensures !$abort ==> ...`.

Procedure Call

```
resource struct T {
    val: u64
}

/* ... */

public fun caller(t_ref: &T) acquires T {
    /* ... */
    callee(t_ref);
    /* ... */
}

spec fun caller {
    modifies global<T>(0x9).val;
}

public fun callee(t_ref: &T) acquires T /* ... */
```

Encoding to Viper

```
method $caller($t_ref: Ref, $T_holders: Set[Ref])
    /* ... */
{
    /* ... */
    inhale acc($t_ref.T$val, read$())
    inhale acc($t_ref.T$val.val_u64, read$()) // new
    var $tmp_T_holders: Set[Ref]
    $tmp_T_holders := $new_T_holders
    exhale address$(9) in $new_T_holders ==> acc(address$(9).T$val.
        val_u64, read$()) // new
    exhale address$(9) in $new_T_holders ==> acc(address$(9).T$val,
        read$())
    inhale forall $r: Ref :: { $r.T$val } $r in $new_T_holders ==> acc(
        $r.T$val, read$())
    inhale forall $r: Ref :: { $r.T$val.val_u64 } $r in $new_T_holders
        ==> acc($r.T$val.val_u64, read$()) // new
    $abort, $new_T_holders := $callee($t_ref, $new_T_holders)
    if ($abort) {
        goto end
    }
    exhale forall $r: Ref :: { $r.T$val.val_u64 } $r in $tmp_T_holders
        ==> acc($r.T$val.val_u64, read$()) // new
    exhale forall $r: Ref :: { $r.T$val } $r in $tmp_T_holders ==> acc(
        $r.T$val, read$())
    inhale address$(9) in $tmp_T_holders ==> acc(address$(9).T$val,
        read$())
    inhale address$(9) in $tmp_T_holders ==> acc(address$(9).T$val.
        val_u64, read$()) // new
    exhale acc($t_ref.T$val.val_u64, read$()) // new
    exhale acc($t_ref.T$val, read$())
    /* ... */
}
```

Figure 4.45: Updated Procedure Call Encoding

MoveTo

```
move_to<Player>(sender, p);
```

Encoding to Viper

```
if ($sender.val_address in $new_Player_holders) {
    $abort := true
    goto end
}
$sender.val_address.Player$strength := $var_5.Player$strength;
// new
inhale (forall $r: Ref ::
    ($r in $new_Player_holders) ==>
    $r.Player$strength != $var_5.Player$strength)

$sender.val_address.Player$alive := $var_5.Player$alive;
// new
inhale (forall $r: Ref ::
    ($r in $new_Player_holders) ==>
    $r.Player$alive != $var_5.Player$alive)

$new_Player_holders := $new_Player_holders
    union Set($sender.val_address)
```

Figure 4.46: Updated Encoding of MoveTo

Abort Conditions

```
spec fun /* ... */ {
    aborts_if !exists<T>(addr);
    aborts_if global<T>(addr).
        val > 99;
}
```

Encoding to Viper

```
method /* ... */
    /* ... */
    ensures $abort <==>
        old(!(addr.val_address
            in $T_holders)) ||
        old(addr.val_address.
            T$val.val_u64 > 99)
```

Figure 4.47: Encoding of `aborts_if`

Postconditions

```
spec fun /* ... */ {
    /* ensures exists<T>(addr);
        */
    ensures global<T>(addr).val
        == 42;
}
```

Encoding to Viper

```
method /* ... */
    /* ... */
    // implicit
    ensures !$abort ==> addr.
        val_address in
        $new_T_holders

    // explicit
    ensures !$abort ==> addr.
        val_address.T$val.
        val_u64 == 42
```

Figure 4.48: Encoding of `ensures`

moves_to

```
1 spec fun /* ... */ {
2     moves_to if(cond) global<T>(Signer::spec_address_of(account))
3 }
```

Encoding to Viper

```
1 method /* ... */
2     /* ... */
3     ensures !$abort ==> (forall $r: Ref ::
4         { ($r in $new_T_holders) }
5         ($r in $new_M$T_holders) ==
6             (($r in $M$T_holders)
7             || (old(cond) && $r == old($account.val_address)))
```

Figure 4.49: Encoding of `moves_to` as postcondition

Self-Framing Postconditions

In order to make the specifications slightly less verbose, we do not require them to be self-framing, i.e. we do not require the developer to explicitly specify that all resources that are referenced in the expression need to exist. Consider the example in Figure 4.48. The `ensures global<T>(addr).val == 42;` specification only makes sense if a resource exists at `addr`. This implies the other postcondition which is commented out on line 2. In our encoding, we include these postconditions necessary to frame other postconditions.

4.4.3 Moves

In Section 4.3, we encoded the permissions associated with `moves_to` and `moves_from` annotations. Now, let us also encode this annotation as a post-condition specifying the new set of resource holders. In order to support move conditions as well as multiple moves of the same resource type to and from accounts, we use quantified permissions, as shown in Figure 4.49.

4.4.4 Specification Functions

In Section 3.4.1, we saw specification functions defined on a module level which enable encapsulation and avoid code duplication. While encoding them as Viper functions may seem like a natural fit, there is a problem with this encoding. In Move procedures, reading the value of a global resource field is a very explicit operation requiring a global borrow which allows Viper to encode the checking of whether the resource exists. Specification functions, in contrast, are non-aborting and the verification, as opposed to the transaction, should fail if access permissions are missing for a resource field. Functions in Viper, however, need to be self-framing, which means

Specification Function

```
spec module {
    define not_initialised(a: address): bool {
        !exists<T>(a) ||
        global<T>(a).val == 0
    }
}
```

Attempt at Encoding to Viper

```
function $not_initialised(a: Ref, $T_holders: Set[Ref])
    requires /* ... */
{
    !(a.val_address in $T_holders) ||
    a.val_address.T$val.val_u64 == 0
}
```

Figure 4.50: Attempt at Encoding Specification Functions as Viper Functions

Specification using Function

```
ensures not_initialised(0x9);
```

Encoding to Viper

```
ensures !(address$(9) in $T_holders) ||
    address$(9).T$val.val_u64 == 0
```

Figure 4.51: Inlined Specification Function

that when translating a Move specification function to a Viper function, we need to ensure that the Viper function's preconditions cover all required read permissions. Consider, for example, Figure 4.50. The `$not_initialised` function requires a read permission to `a.val_address.T$val.val_u64`, however, whether we have this permission depends on whether the resource exists. Encoding such access permission conditions is difficult and we would essentially specify the function body in its preconditions. Instead, we will inline specification functions, as shown in Figure 4.51. This, however, means that recursion in specification functions is not supported.

4.4.5 Global Properties

The last part of the specification language we need to encode is global property specifications. We will first look at the encoding of invariants and then encode special global property functions.

Module Invariant

```
/* ... */
module invariant exists<T>(0x9) ==> global<T>(0x9).val > 0;
/* ... */

public fun foo /* ... */
```

Encoding to Viper

```
method $invariant($T_holders: Set[Ref])
    requires |$T_holders| == 0
{
    assert address$(9).val_address in $T_holders
        ==> address$(9).val_address.T$val.val_u64 > 0
}

method $foo /* ... */
    /* ... */
    requires address$(9).val_address in $T_holders
        ==> address$(9).val_address.T$val.val_u64 > 0
    /* ... */
    ensures address$(9).val_address in $new_T_holders
        ==> address$(9).val_address.T$val.val_u64 > 0
```

Figure 4.52: Module Invariant Encoding

Count Operation

```
count<Player>()
```

Encoding to Viper

```
|$Player_holders|
```

Figure 4.53: Encoding of `count`

Module Invariants

Recall that a module invariant is an expression that needs to hold in every possible global state that is permitted by the module procedures. This means that we first need to establish that the invariant holds in the initial state where no procedure is called and all resource holder sets are empty. Then, we may assume that the invariant holds before every public procedure call and we need to verify that it still holds after. This is shown in Figure 4.52.

Global Resource Functions

In Section 3.2.2, we saw special `count` and `sum` operations that allow us to reason about properties over all account resources of one type. For both these operations and similar ones that may be used in the future, we can use the resource holder set for a natural encoding. While the encoding of `count` is very straightforward, as shown in Figure 4.53, for the encoding of `sum` we define an additional function for every field that we sum over. Since

Count Operation

```
public fun init(account: &signer) {
    move_to(account, Player {
        strength: 50,
        alive: true
    });
}

spec fun init {
    /* ... */
    ensures sum<Player>().strength ==
        old(sum<Player>().strength) + 50
}
```

Encoding to Viper

```
function sum$Player$strength($Player_holders): Int
    requires forall $r ::
        { $r in $Player_holders }
        ($r in $Player_holders) ==>
        acc($r.Player$strength, read$())
    requires forall $r ::
        { $r in $Player_holders }
        ($r in $Player_holders) ==>
        acc($r.Player$strength.val_u64, read$())
    ensures |$Player_holders| == 0 ==> result == 0

method $init($account: Ref, $Player_holders: Set[Ref]) /* ... */
    /* ... */
    ensures !$abort ==> sum$Player$strength($new_Player_holders)
        == old(sum$Player$strength($Player_holders)) + 50
{
    /* ... */
    $tmp_Player_holders := $new_Player_holders
    $new_Player_holders := $new_Player_holders union Set($var_3)
    inhale sum$Player$strength($new_Player_holders) ==
        sum$Player$strength($tmp_Player_holders) +
        $var_3.Player$strength.val_u64
    /* ... */
}
```

Figure 4.54: Encoding of `sum`

defining a closed expression for the sum of fields of `Ref`s is hard to achieve in Viper, we simply update the value of the function whenever the set of resource holders is updated. An example can be seen in Figure 4.54.

Chapter 5

Implementation and Evaluation

In this chapter, we will describe how we implemented our prototype verifier, evaluate its strengths and weaknesses and discuss which lessons can be drawn.

5.1 Verifier Structure

In order to ensure easy integration with the Libra code base and in order to be able to use Prusti's Viper-encoding libraries, both of which are developed in Rust, we implemented our verifier in Rust. When verifying a Move program, which may contain one or more modules, we first translate the Move program to Move bytecode using the Move compiler, which is then passed to the stackless bytecode generator developed for the Move Prover, which replaces stack operations by variable operations and returns stackless bytecode. This process does not discard program annotations but stores both stackless bytecode and specifications in an environment variable. In order to support our specification language, we added the additional keywords used by our specification language to a forked version of the Libra repository.

Next, we translate the Move modules which are stored in the environment

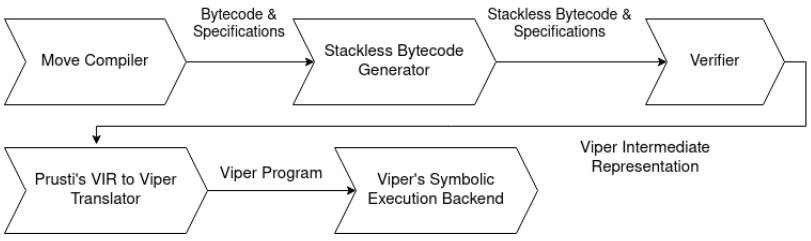

Figure 5.1: Translation of a Move Program to a Viper Program

variable. Recall that modules may depend on each other but must not exhibit cyclic dependencies, so we begin by translating the modules that have no untranslated dependencies. For each module, we first encode resources as fields, then specification functions and lastly procedures. While our verifier supports both inlining of specification functions and translating them to Viper functions, we currently always default to inlining in order to solve the problem described in Section 4.4.4. This means that we currently do not support recursive specification functions. Since specification functions may call one another, and these calls need to be inlined, we translate the specification functions in order of dependency. In contrast, we support the encoding of recursive procedures. In order to support procedure calls, we first store the information required to make a procedure call for every procedure, before translating the procedure's instructions and specifications. In doing so, we build up control flow graphs encoded to the Viper intermediate representation, which are then translated to a Viper program. Lastly, this Viper program is verified by the Viper's symbolic execution backend.

5.2 Supported Parts of the Move Language

Our verifier supports the specification language we described in Chapter 3 as well as all bytecode instructions described in Chapter 4 with the exception of the implementation of the `move_from` instruction and of nested resources. The instructions implemented cover all essential parts of the Move language even as some arithmetic and logical operations are not implemented due to time constraints. In principle, our encoding allows for the verification of all Move instructions and operations. The specification language we described in Chapter 3 is far less extensive than the specification language used by the Move Prover but can describe many interesting properties of Move programs. Like the Move Prover, our specification language does not yet support reasoning about unbounded loops. Additionally, in the future, our specification language will require more constructs like `count` and `sum` to reason about global resource properties.

With the exception of the check that module invariants hold in the initial state, which has not yet been implemented, the verifier we have built is sound and, in particular, soundly encodes framing.

Currently, the key restriction of our verifier is the lack of support for native vector functions. In order to be able to verify the Move standard library, Move's vector operations would need to be encoded to Viper. Since vectors come up throughout the standard library, we can currently only verify parts of the standard library, such as the `Timestamp` module.

Move Program	Verification Time
Aliasing Move and Modifies	1 min 30 sec
Standard Library Timestamp	3 min
Referendum Module	25 min
Game Module	90 min
Game Transaction	7 hours

Table 5.1: Verification times for a number of programs

5.3 Performance Evaluation

While our verifier can fully capture and reason about changes to the global state caused by complex transactions, the verification process is currently slow for complex programs. While the verification of simple programs, where there are few global resources handled and no potential aliasing issues, is completed within seconds, complex programs can take several hours to verify. This is likely caused by the prevalence of quantified permissions. For every resource acquired by a procedure, we currently have three quantified preconditions and postconditions per resource field. In addition, to establish a frame, we may need to inhale and exhale numerous quantified permissions if the procedure called acquires a large number of resources. We also use quantified permissions to encode the possible aliasing of addresses where resources are modified. In combination, these quantified expressions have the potential to slow down verification significantly. Table 5.1 shows the approximate verification times required for a number of Move programs on a 16GB RAM, 8 core i7 architecture running Linux. The Move programs referenced are shown in the appendix. Simpler code examples like the ones given throughout the thesis, typically verify in less than 10 seconds.

5.4 Discussion of our Approach

The way the Move language is designed, procedures are generally allowed to read from and write to an unlimited number of locations, provided they specify the respective resource in the `acquires` list. This fundamentally makes reasoning about the exact changes caused by a procedure difficult and, in general case, intractable. While reading arbitrary locations does not cause issues in the general case, in order to support modular verification, we require the exact locations that a procedure may modify. We solved this problem by introducing `modifies` annotations. Using Viper's permissions-based model, we have a natural way to encode these annotations and handle issues of aliasing. While Viper may support other ways of encoding the read permissions required by procedures than quantified permissions, we felt they were a natural fit. While resource field references complicate these quantified permissions, our usage of quantified permissions enables us to

implicitly encode a procedure's frame in an automatable way. The framing problem may be particularly hard to solve without quantified permissions, but there is a possibility that there exists a simpler and faster modular encoding with far fewer quantified permissions. Currently, the speed restrictions caused by quantified permissions make our verifier impractical for complex modules and transactions. Optimisations of the encoding as well as optimisations of quantified permissions on a Viper level could sufficiently speed up the verification process to make it useful in practice.

5.4.1 Comparison to the Move Prover

While the Move Prover's verification process is currently not fully modular, it is powerful enough to encode all relevant properties of a Move program and it is fast enough to be an effective tool in practice. The Move Prover may potentially struggle with large transactions that make many calls to various modules, but the way the Move language is designed, we expect transactions to remain reasonably small. Currently, the Move Prover does not generally support recursive procedures and it will be interesting to see how this problem will be solved.

5.5 Possible Changes to the Move Language

As part of the process of designing a Move verifier, we came across a number of points where new features or restrictions on the Move language might simplify verification.

- In general, it would greatly help the verification process if complete framing information were available from a procedure's signature. While the `acquires` list provides some of this information, critically it does not specify which addresses will be modified. While we added this information to procedure specifications, requiring this information in the Move language itself would allow far better reasoning about procedures that are not annotated with specifications. In particular, we could establish a frame for these procedures which may be sufficient for reasoning about their effects when called from another procedure. One could, for example, imagine an extended `acquires` syntax of the form `acquires T[addr1_modified, addr2_modified]` and `moves_to` and `moves_from` keywords.
- Alternatively, verifiers could make an attempt at statically analysing at which account addresses resources are moved and modified. Part of the problem with this is the possibility of borrowing from addresses that are stored inside resources. In many cases, these addresses will only be used for comparison purposes as, for example, in the standard library's `Offer` module. Indicating in the Move language whether

an address may be used in a `borrow_global_mut` or `move_from` instruction could help a verifier establish which addresses are in fact modified. One possible syntax could be `non_acquirable_field: address` vs. `acquirable_field: address*`.

- In our encoding, which relies heavily on quantified permissions, references to resource fields introduce an extra level of Viper fields that we require quantified permissions as well as an injectivity condition for. While this issue may be specific to our encoding, references are generally difficult to reason about. The Move language could be simplified by not supporting field references and instead requiring the developer to specify the resource field directly as in `res.field`.

Chapter 6

Conclusion and Future Work

The goal of this project was to design a specification language and an encoding to Viper for the Move language and to implement the encoding in a prototype verifier. We have managed to achieve this goal and have developed a modular verifier that can reason about the exact changes made by Move transactions as well as non-trivial global properties and invariants. We achieved this by exploiting the information provided by Move's type system, which is inspired by the Rust programming language.

As Prusti has successfully allowed easy-to-use verification of complex Rust programs, we suspected that Viper might be a good fit for the Move language as well. While we were able to directly use some of the lessons learned from Prusti and were, for example, able to encode aliasing guarantees in a natural way, the semantics of Move resources is markedly different in some ways to Rust programs. In particular, Move procedures are allowed to read and modify resources at an arbitrary number of accounts with little information provided about the exact behaviour by the procedure signature. To establish procedure frames, we therefore required additional specifications, making the verification process less automated than in the case of Rust.

To account for the various possible resources that procedures have read access to, our encoding makes use of numerous quantified expressions and permissions. The verification of Viper programs containing many quantified permissions turned out to currently be very slow, making our verifier impractical for complex transactions.

The Move Prover, in contrast, is currently in part not modular and, therefore, does not require exact frame information and complex quantified permissions. In theory, this approach should be quite slow for very complex transaction; however, the way the Move language is designed, such transactions are likely to be uncommon, and for the small amount of Move code that currently exists, the Move Prover is efficient enough to be very useful

in practice.

Two main challenges remain for the future. Firstly, while all important Move instructions have a Viper encoding, our support for native Move functions is currently very limited. In particular, our verifier needs to support vectors in order to verify substantial parts of the Move standard library. The second challenge is to optimise the verification process to make it more useful in practice. This will probably involve optimisations of the encoding itself and the optimisation of quantified permissions in Viper.

Appendix

Listing 6.1: Aliasing Move and Modifies Addresses

```
1  module M {
2      use 0x1::Signer;
3
4      resource struct T {
5          f: address
6      }
7
8      resource struct S {
9          h: address
10     }
11
12     resource struct U {
13         g: u64
14     }
15
16     public fun foo(account: &signer, addr1: address, addr2: address)
           acquires S, T, U {
17         bar(account, addr1, addr2);
18         bar2(account);
19     }
20
21     spec fun foo {
22         aborts_if exists<U>(Signer::spec_address_of(account));
23
24         aborts_if !exists<T>(addr1);
25         aborts_if !exists<U>(global<T>(addr1).f) && global<T>(addr1).f
               != Signer::spec_address_of(account);
26
27         aborts_if !exists<S>(addr2);
28         aborts_if !exists<U>(global<S>(addr2).h) && global<S>(addr2).h
               != Signer::spec_address_of(account);
29
30         aborts_if !exists<U>(Signer::spec_address_of(account));
31
32         moves_to global<U>(Signer::spec_address_of(account));
33
34         modifies global<U>(global<T>(addr1).f).g;
35         modifies global<U>(global<S>(addr2).h).g;
36         modifies global<U>(Signer::spec_address_of(account)).g;
37
38         ensures global<U>(Signer::spec_address_of(account)).g == 3;
```

```
39      }
40
41      public fun bar(account: &signer, addr1: address, addr2: address)
            acquires S, T, U {
42          move_to<U>(account, U {
43              g: 0
44          });
45
46          borrow_global_mut<U>(borrow_global<T>(addr1).f).g = 1;
47          borrow_global_mut<U>(borrow_global<S>(addr2).h).g = 2;
48      }
49
50      spec fun bar {
51          aborts_if exists<U>(Signer::spec_address_of(account));
52
53          aborts_if !exists<T>(addr1);
54          aborts_if !exists<U>(global<T>(addr1).f) && global<T>(addr1).f
                != Signer::spec_address_of(account);
55
56          aborts_if !exists<S>(addr2);
57          aborts_if !exists<U>(global<S>(addr2).h) && global<S>(addr2).h
                != Signer::spec_address_of(account);
58
59          moves_to global<U>(Signer::spec_address_of(account));
60
61          modifies global<U>(global<T>(addr1).f).g;
62          modifies global<U>(global<S>(addr2).h).g;
63
64          ensures global<U>(Signer::spec_address_of(account)).g == 0
65              || global<U>(Signer::spec_address_of(account)).g == 1
66              || global<U>(Signer::spec_address_of(account)).g == 2;
67      }
68
69      fun bar2(account: &signer) acquires U {
70          borrow_global_mut<U>(Signer::address_of(account)).g = 3;
71      }
72
73      spec fun bar2 {
74          modifies global<U>(Signer::spec_address_of(account));
75          aborts_if !exists<U>(Signer::spec_address_of(account));
76          ensures global<U>(Signer::spec_address_of(account)).g == 3;
77      }
78  }
```

Listing 6.2: Referendum Module

```
1   module Referendum {
2       use 0x1::Signer;
3
4       spec module {
5           invariant forall a: address : a != 0x12345678 ==> !exists<
                Counter>(a);
6           invariant count<VoteToken>() > 0 ==> exists<Counter>(0x12345678)
                ;
7           invariant count<VoteToken>() == 0 ==> !exists<Counter>(0
                x12345678) ||
8               (global<Counter>(0x12345678).in_favour == 0 && global<
                    Counter>(0x12345678).against == 0);
9           invariant count<VoteToken>() > 0 ==>
10              count<VoteToken>() ==
11                  global<Counter>(0x12345678).in_favour +
12                  global<Counter>(0x12345678).against;
13      }
```

```
14
15      resource struct Counter {
16          in_favour: u64,
17          against: u64
18      }
19
20      resource struct VoteToken {}
21
22      public fun init(account: &signer) {
23          if (Signer::address_of(account) != 0x12345678) {
24              abort 100
25          };
26          move_to<Counter>(account, Counter {
27              in_favour: 0,
28              against: 0
29          });
30      }
31
32      spec fun init {
33          moves_to global<Counter>(0x12345678);
34          aborts_if Signer::spec_address_of(account) != 0x12345678;
35          aborts_if exists<Counter>(0x12345678);
36          ensures exists<Counter>(0x12345678);
37          ensures global<Counter>(0x12345678).in_favour == 0;
38          ensures global<Counter>(0x12345678).against == 0;
39      }
40
41      public fun vote(account: &signer, in_favour: bool) acquires Counter
            {
42          let counter_ref = borrow_global_mut<Counter>(0x12345678);
43          if (in_favour) {
44              counter_ref.in_favour = counter_ref.in_favour + 1;
45          } else {
46              counter_ref.against = counter_ref.against + 1;
47          };
48          move_to<VoteToken>(account, VoteToken { });
49      }
50
51      spec fun vote {
52          moves_to global<VoteToken>(Signer::spec_address_of(account));
53          modifies global<Counter>(0x12345678);
54          aborts_if !exists<Counter>(0x12345678);
55          aborts_if exists<VoteToken>(Signer::spec_address_of(account));
56          aborts_if in_favour && global<Counter>(0x12345678).in_favour ==
                max_u64();
57          aborts_if !in_favour && global<Counter>(0x12345678).against ==
                max_u64();
58          ensures exists<Counter>(0x12345678);
59          ensures in_favour ==>
60              global<Counter>(0x12345678).in_favour ==
61              old(global<Counter>(0x12345678).in_favour) + 1;
62          ensures in_favour ==>
63              global<Counter>(0x12345678).against ==
64              old(global<Counter>(0x12345678).against);
65          ensures !in_favour ==>
66              global<Counter>(0x12345678).against ==
67              old(global<Counter>(0x12345678).against) + 1;
68          ensures !in_favour ==>
69              global<Counter>(0x12345678).in_favour ==
70              old(global<Counter>(0x12345678).in_favour);
71      }
72  }
```

Listing 6.3: Standard Library Timestamp

```
address 0x1 {

/// This module keeps a global wall clock that stores the current Unix time in microseconds.
/// It interacts with the other modules in the following ways:
///
/// * Genesis: to initialize the timestamp
/// * VASP: to keep track of when credentials expire
/// * LibraSystem, LibraAccount, LibraConfig: to check if the current state is in the genesis state
/// * LibraBlock: to reach consensus on the global wall clock time
/// * AccountLimits: to limit the time of account limits
/// * LibraTransactionTimeout: to determine whether a transaction is still valid
///
module LibraTimestamp {
    use 0x1::CoreAddresses;
    use 0x1::Signer;

    /// A singleton resource holding the current Unix time in microseconds
    resource struct CurrentTimeMicroseconds {
        microseconds: u64,
    }

    /// A singleton resource used to determine whether time has started. This
    /// is called at the end of genesis.
    resource struct TimeHasStarted {}

    const EINVALID_SINGLETON_ADDRESS: u64 = 0;
    const ETIME_NOT_INITIALIZED: u64 = 1;
    const ENOT_VM: u64 = 2;
    const EINVALID_TIMESTAMP: u64 = 3;

    /// Initializes the global wall clock time resource. This can only be called from genesis.
    public fun initialize(lr_account: &signer) {
        // Operational constraint, only callable by the libra root account
        assert(Signer::address_of(lr_account) == CoreAddresses::LIBRA_ROOT_ADDRESS(), EINVALID_SINGLETON_ADDRESS);

        let timer = CurrentTimeMicroseconds { microseconds: 0 };
        move_to(lr_account, timer);
    }

    /// Marks that time has started and genesis has finished. This can only be called from genesis.
    public fun set_time_has_started(lr_account: &signer) acquires CurrentTimeMicroseconds {
        assert(Signer::address_of(lr_account) == CoreAddresses::LIBRA_ROOT_ADDRESS(), EINVALID_SINGLETON_ADDRESS);

        // Current time must have been initialized.
        assert(
            exists<CurrentTimeMicroseconds>(CoreAddresses::LIBRA_ROOT_ADDRESS()) && now_microseconds() == 0,
            ETIME_NOT_INITIALIZED
        );
        move_to(lr_account, TimeHasStarted{});
```

```
50      }
51
52      /// Helper functions for tests to reset the time-has-started, and
            pretend to be in genesis.
53      /*
54      public fun reset_time_has_started_for_test() acquires TimeHasStarted
            {
55          let TimeHasStarted{} = move_from<TimeHasStarted>(CoreAddresses::
                LIBRA_ROOT_ADDRESS());
56      }
57      */
58
59      /// Updates the wall clock time by consensus. Requires VM privilege
            and will be invoked during block prologue.
60      public fun update_global_time(
61          account: &signer,
62          proposer: address,
63          timestamp: u64
64      ) acquires CurrentTimeMicroseconds {
65          // Can only be invoked by LibraVM privilege.
66          assert(Signer::address_of(account) == CoreAddresses::
                VM_RESERVED_ADDRESS(), ENOT_VM);
67
68          let global_timer = borrow_global_mut<CurrentTimeMicroseconds>(
                CoreAddresses::LIBRA_ROOT_ADDRESS());
69          if (proposer == CoreAddresses::VM_RESERVED_ADDRESS()) {
70              // NIL block with null address as proposer. Timestamp must
                    be equal.
71              assert(timestamp == global_timer.microseconds,
                    EINVALID_TIMESTAMP);
72          } else {
73              // Normal block. Time must advance
74              assert(global_timer.microseconds < timestamp,
                    EINVALID_TIMESTAMP);
75          };
76          global_timer.microseconds = timestamp;
77      }
78
79      /// Gets the timestamp representing `now` in microseconds.
80      public fun now_microseconds(): u64 acquires CurrentTimeMicroseconds
            {
81          borrow_global<CurrentTimeMicroseconds>(CoreAddresses::
                LIBRA_ROOT_ADDRESS()).microseconds
82      }
83
84      /// Helper function to determine if the blockchain is in genesis
            state.
85      public fun is_genesis(): bool {
86          !exists<TimeHasStarted>(CoreAddresses::LIBRA_ROOT_ADDRESS())
87      }
88
89      /// Helper function to determine whether the CurrentTime has been
            initialized.
90      public fun is_not_initialized(): bool acquires
            CurrentTimeMicroseconds {
91         !exists<CurrentTimeMicroseconds>(CoreAddresses::
               LIBRA_ROOT_ADDRESS()) || now_microseconds() == 0
92      }
93
94      // **************** GLOBAL SPECIFICATION ****************
95
96      /// # Module specification
```

```
97
98      spec module {
99          /// Verify all functions in this module.
100         pragma verify = true;
101
102         /// Specification version of the `Self::is_genesis` function.
103         define spec_is_genesis(): bool {
104             !exists<TimeHasStarted>(CoreAddresses::
                    SPEC_LIBRA_ROOT_ADDRESS())
105         }
106
107         /// Specification version of the `Self::is_not_initialized`
                function.
108         define spec_is_not_initialized(): bool {
109             !spec_root_ctm_initialized() || spec_now_microseconds() == 0
110         }
111
112         /// True if the association root account has a
                CurrentTimeMicroseconds.
113         define spec_root_ctm_initialized(): bool {
114             exists<CurrentTimeMicroseconds>(CoreAddresses::
                    SPEC_LIBRA_ROOT_ADDRESS())
115         }
116
117         /// Auxiliary function to get the association's Unix time in
                microseconds.
118         define spec_now_microseconds(): u64 {
119             global<CurrentTimeMicroseconds>(CoreAddresses::
                    SPEC_LIBRA_ROOT_ADDRESS()).microseconds
120         }
121     }
122
123     /// ## Persistence of Initialization
124
125     spec schema InitializationPersists {
126         /// If the `TimeHasStarted` resource is initialized and we
                finished genesis, we can never enter genesis again.
127         /// Note that this is an important safety property since during
                genesis, we are allowed to perform certain
128         /// operations which should never be allowed in normal on-chain
                execution.
129         ensures old(!spec_is_genesis()) ==> !spec_is_genesis();
130
131         /// If the `CurrentTimeMicroseconds` resource is initialized, it
                 stays initialized.
132         ensures old(spec_root_ctm_initialized()) ==>
                spec_root_ctm_initialized();
133     }
134
135     spec module {
136         apply InitializationPersists to * except
                reset_time_has_started_for_test;
137     }
138
139     /// ## Global Clock Time Progression
140
141     spec schema GlobalWallClockIsMonotonic {
142         /// The global wall clock time never decreases.
143         ensures old(spec_root_ctm_initialized()) ==> (old(
                spec_now_microseconds()) <= spec_now_microseconds());
144     }
145     spec module {
```

```
146         apply GlobalWallClockIsMonotonic to *;
147     }
148
149     // **************** FUNCTION SPECIFICATIONS ***************
150
151     spec fun initialize {
152         /* added */ moves_to global<CurrentTimeMicroseconds>(Signer::
                spec_address_of(lr_account));
153         aborts_if Signer::spec_address_of(lr_account) != CoreAddresses::
                SPEC_LIBRA_ROOT_ADDRESS();
154         aborts_if spec_root_ctm_initialized();
155         ensures spec_root_ctm_initialized();
156         ensures spec_now_microseconds() == 0;
157     }
158
159     spec fun set_time_has_started {
160         /* added */ moves_to global<TimeHasStarted>(Signer::
                spec_address_of(lr_account));
161         aborts_if Signer::spec_address_of(lr_account) != CoreAddresses::
                SPEC_LIBRA_ROOT_ADDRESS();
162         aborts_if !spec_is_genesis();
163         aborts_if !spec_root_ctm_initialized();
164         aborts_if spec_now_microseconds() != 0;
165         ensures !spec_is_genesis();
166     }
167
168     spec fun update_global_time {
169         /* added */ modifies global<CurrentTimeMicroseconds>(
                CoreAddresses::SPEC_LIBRA_ROOT_ADDRESS());
170         pragma assume_no_abort_from_here = true;
171         aborts_if Signer::spec_address_of(account) != CoreAddresses::
                SPEC_VM_RESERVED_ADDRESS();
172         aborts_if !spec_root_ctm_initialized();
173         aborts_if (proposer == CoreAddresses::SPEC_VM_RESERVED_ADDRESS()
                ) && (timestamp != spec_now_microseconds());
174         aborts_if (proposer != CoreAddresses::SPEC_VM_RESERVED_ADDRESS()
                ) && !(timestamp > spec_now_microseconds());
175         ensures spec_now_microseconds() == timestamp;
176     }
177
178     spec fun now_microseconds {
179         /* added */ ensures result == spec_now_microseconds();
180         include TimeAccessAbortsIf;
181     }
182
183     spec fun is_genesis {
184         aborts_if false;
185         ensures result == spec_is_genesis();
186     }
187
188     spec fun is_not_initialized {
189         aborts_if false;
190         ensures result == spec_is_not_initialized();
191     }
192
193     spec schema TimeAccessAbortsIf {
194         aborts_if !exists<CurrentTimeMicroseconds>(CoreAddresses::
                SPEC_LIBRA_ROOT_ADDRESS());
195     }
196 }
197
198 }
```

Listing 6.4: Game Module

```
module Game {
    use 0x1::Signer;

    resource struct Player {
        strength: u64,
        alive: bool
    }

    // donates the donor's strength to the beneficiary
    public fun donate(donor: &signer, beneficiary: address) acquires
        Player {
        // read donor strength
        let donor_ref: &Player = borrow_global<Player>(Signer::
            address_of(donor));
        let donor_strength = get_strength(donor_ref);

        // read beneficiary strength
        let beneficiary_ref: &Player = borrow_global<Player>(beneficiary
            );
        let beneficiary_strength = get_strength(beneficiary_ref);

        // calculate new strength
        let new_strength = donor_strength + beneficiary_strength;

        // set beneficiary strength to new strength
        let mut_beneficiary_ref: &mut Player = borrow_global_mut<Player
            >(beneficiary);
        mut_beneficiary_ref.strength = new_strength;

        // set donor strength to zero
        let mut_donor_ref: &mut Player = borrow_global_mut<Player>(
            Signer::address_of(donor));
        mut_donor_ref.strength = 0;
    }

    spec fun donate {
        modifies global<Player>(Signer::spec_address_of(donor)).strength
            ;
        modifies global<Player>(beneficiary).strength;

        aborts_if !exists<Player>(Signer::spec_address_of(donor));
        aborts_if !exists<Player>(beneficiary);
        aborts_if global<Player>(Signer::spec_address_of(donor)).
            strength + global<Player>(beneficiary).strength > max_u64();

        ensures Signer::spec_address_of(donor) != beneficiary ==>
            (global<Player>(Signer::spec_address_of(donor)).strength ==
                0 &&
             global<Player>(beneficiary).strength == old(global<Player>(
                 beneficiary).strength)
                + old(global<Player>(Signer::spec_address_of(donor)).
                    strength));
        ensures Signer::spec_address_of(donor) == beneficiary ==>
            (global<Player>(Signer::spec_address_of(donor)).strength ==
                0);
        ensures sum<Player>().strength <= old(sum<Player>().strength);
    }

    fun get_strength(player: &Player): u64 {
        player.strength
    }
```

```
51
52      spec fun get_strength {
53          ensures result == player.strength;
54      }
55  }
```

Listing 6.5: Game Transaction

```
1   module Transaction {
2       use 0x1::Signer;
3       use 0x789::Coin;
4       use 0x789::Game;
5
6       public fun main1(account: &signer) {
7           Coin::init(account);
8           Game::join_game(account);
9       }
10
11      spec fun main1 {
12          moves_to global<Coin::T>(Signer::spec_address_of(account));
13          moves_to global<Game::PlayerState>(Signer::spec_address_of(
                account));
14          modifies global<Coin::T>(Game::owner());
15          aborts_if true;
16      }
17  }
18
19  module Game {
20      use 0x1::Signer;
21      use 0x789::Coin;
22
23      spec module {
24          define owner(): address {
25              0x12345
26          }
27
28          define sender(account: signer): address {
29              Signer::spec_address_of(account)
30          }
31      }
32
33      resource struct PlayerState {
34          alive: bool,
35          strength: u64
36      }
37
38      public fun join_game(account: &signer) {
39          // Without this check, the postconditions are incorrect
40          if (Signer::address_of(account) == 0x12345) {
41              abort 100
42          };
43
44          let coin = Coin::withdraw(account, 50);
45          Coin::deposit(0x12345, coin);
46          move_to<PlayerState>(account, PlayerState {
47              alive: true,
48              strength: 100
49          })
50      }
51
52      spec fun join_game {
53          aborts_if Signer::spec_address_of(account) == owner();
54          moves_to global<PlayerState>(sender(account));
```

```
55          modifies global<Coin::T>(sender(account));
56          modifies global<Coin::T>(owner());
57          aborts_if !exists<Coin::T>(sender(account));
58          aborts_if Coin::value(global<Coin::T>(sender(account))) < 50;
59          aborts_if !exists<Coin::T>(owner());
60          aborts_if Coin::value(global<Coin::T>(owner())) > max_u64() -
                50;
61          aborts_if exists<PlayerState>(sender(account));
62          ensures exists<Coin::T>(sender(account));
63          ensures exists<Coin::T>(owner());
64          ensures exists<PlayerState>(sender(account));
65          ensures Coin::value(global<Coin::T>(owner())) == old(Coin::value
                (global<Coin::T>(owner()))) + 50;
66          ensures Coin::value(global<Coin::T>(sender(account))) == old(
                Coin::value(global<Coin::T>(sender(account)))) - 50;
67          ensures global<PlayerState>(sender(account)).alive;
68          ensures global<PlayerState>(sender(account)).strength == 100;
69      }
70
71      public fun donate_strength(account: &signer, beneficiary: address)
            acquires PlayerState {
72          // Without this check, the postconditions are incorrect
73          if (Signer::address_of(account) == beneficiary) {
74              abort 100
75          };
76
77          let old_strength = borrow_global<PlayerState>(Signer::address_of
                (account)).strength;
78          let ben_strength_ref = &mut borrow_global_mut<PlayerState>(
                beneficiary).strength;
79          *ben_strength_ref = *ben_strength_ref + old_strength;
80          let sender_strength_ref = &mut borrow_global_mut<PlayerState>(
                Signer::address_of(account)).strength;
81          *sender_strength_ref = 0;
82      }
83
84      spec fun donate_strength {
85          aborts_if Signer::spec_address_of(account) == beneficiary;
86          modifies global<PlayerState>(Signer::spec_address_of(account)).
                strength;
87          modifies global<PlayerState>(beneficiary).strength;
88          aborts_if !exists<PlayerState>(Signer::spec_address_of(account))
                ;
89          aborts_if !exists<PlayerState>(beneficiary);
90          aborts_if global<PlayerState>(Signer::spec_address_of(account)).
                strength + global<PlayerState>(beneficiary).strength >
                max_u64();
91          ensures exists<PlayerState>(Signer::spec_address_of(account));
92          ensures exists<PlayerState>(beneficiary);
93          ensures global<PlayerState>(Signer::spec_address_of(account)).
                strength == 0;
94          ensures global<PlayerState>(beneficiary).strength
95              == old(global<PlayerState>(beneficiary).strength)
96              + old(global<PlayerState>(Signer::spec_address_of(account)).
                    strength);
97      }
98  }
99
100 module Coin {
101     use 0x1::Signer;
102
103     resource struct T {
```

```
104         amount: u64
105     }
106
107     public fun zero(): T {
108         T {
109             amount: 0
110         }
111     }
112
113     spec fun zero {
114         aborts_if false;
115         ensures value(result) == 0;
116     }
117
118     public fun init(account: &signer) {
119         move_to<T>(account, T {
120             amount: 0
121         });
122     }
123
124     spec fun init {
125         aborts_if exists<T>(Signer::spec_address_of(account));
126         moves_to global<T>(Signer::spec_address_of(account));
127         ensures global<T>(Signer::spec_address_of(account)).amount == 0;
128     }
129
130     public fun withdraw(account: &signer, amount: u64): T acquires T {
131         let t_ref = borrow_global_mut<T>(Signer::address_of(account));
132         if (t_ref.amount < amount) {
133             abort 100
134         };
135         t_ref.amount = t_ref.amount - amount;
136         T {
137             amount: amount
138         }
139     }
140
141     spec fun withdraw {
142         modifies global<T>(Signer::spec_address_of(account)).amount;
143         aborts_if !exists<T>(Signer::spec_address_of(account));
144         aborts_if value(global<T>(Signer::spec_address_of(account))) <
                amount;
145         ensures exists<T>(Signer::spec_address_of(account));
146         ensures value(global<T>(Signer::spec_address_of(account))) ==
                old(value(global<T>(Signer::spec_address_of(account)))) -
                amount;
147         ensures value(result) == amount;
148     }
149
150     public fun deposit(receiver: address, coin: T) acquires T {
151         let T { amount } = coin;
152         let t_ref = borrow_global_mut<T>(receiver);
153         t_ref.amount = t_ref.amount + amount;
154     }
155
156     spec fun deposit {
157         modifies global<T>(receiver).amount;
158         aborts_if !exists<T>(receiver);
159         aborts_if value(global<T>(receiver)) + value(coin) > max_u64();
160         ensures value(global<T>(receiver)) == old(value(global<T>(
                receiver))) + value(coin);
161     }
```

```
162 }
```

Bibliography

[1] Libra Association. Libra White Paper. https://libra.org/en-US/white-paper, 2020.

[2] Libra Association. Move: A Language With Programmable Resources. https://developers.libra.org/docs/assets/papers/libra-move-a-language-with-programmable-resources/2020-05-26.pdf, 2020.

[3] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. Leveraging rust types for modular specification and verification. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019.

[4] O'Hearn P. Reynolds J. Yang H. Local reasoning about programs that alter data structures. In: Fribourg, L. (ed.) CSL 2001 and EACSL 2001. LNCS, vol. 2142 Springer, Heidelberg, 2001.

[5] M. Leino K. Rustan. This is Boogie 2. https://www.microsoft.com/en-us/research/wp-content/uploads/2016/12/krml178.pdf, 2008.

[6] S. Drossopoulou. Libra F. Schrans, S. Eisenbach. Writing Safe Smart Contracts in Flint. https://dl.acm.org/doi/10.1145/3191697.3213790, 2018.

[7] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. *Viper: A Verification Infrastructure for Permission-Based Reasoning*, pages 41–62. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.

[8] M. Bartoletti N. Atzei and T. Cimoli. A Survey of Attacks on Ethereum Smart Contracts (SoK). *International Conference on Principles of Security and Trust*, 2017.

[9] Jingyi Zhong, Kevin Cheang, Shaz Qadeer, Wolfgang Grieskamp, Sam Blackshear, Junkil Park, Yoni Zohar, Clark Barrett, and David Dill. The Move Prover. https://link.springer.com/content/pdf/10.1007%2F978-3-030-53288-8_7.pdf, 2020.

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Verification of Programs Written in Libra's Move Language

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Müller	Constantin

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
London, 02.09.2020	Constantin Müller

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.