Port Move to WebAssembly

yuyangxxxmail@gmail.com

June 2, 2023

1 Introduction

1.1 Introduce to Move

Move is a programming language for writing safe smart contracts originally developed at Facebook to power the Diem blockchain.

Move is a strongly-statically-typed programming language originally developed at Facebook for smart contract, Right now Move mainly use in 'Sui' and 'Aptos'.

Move is written in Rust and is heavily influenced by Rust,Move followed the Rust ownership system.

A example of Move code.

```
 module 0x42::test {
 struct Example has copy, drop { i: u64 }

 use std::debug;
 friend 0x42::another_test;

 const ONE: u64 = 1;

 public fun print(x: u64) {
     let sum = x + ONE;
     let example = Example { i: sum };
     debug::print(&sum)
 }
}
```

Move code is compile to Move bytecode to run, before run There is a bytecode verifier ensure that the Move bytecode is well formed.

1.2 Introduce to WebAssembly

WebAssembly (abbreviated Wasm) is a binary instruction format for a stack-based virtual machine. Wasm is designed as a portable compilation target for programming languages, enabling deployment on the web for client and server applications.

WebAssembly is a neutral universal compile target, It is very easy from high level language to generate.

Stack-based virtual machine is a machine that have *operand stack* and operation that only operate the value on stack.

Here is a example.

```
local.get 0 // push local variable 0 to stack
local.get 1 // push local variable 1 to stack
i32.add     // pop out two value on stack and perform addition
```

Another thing that I want emphasize is that wasm doesn't have a *goto* statement.

After many years of practise of programming, The industry founds *goto* is very harmful.

wasm designed 'loop' 'if' ,etc to get rid of harmful *goto*.
For examples.

```
loop
   local.get 0
   local.get 0
   break  // break loop
   ...
   end // end of the loop
```

These key features make wasm is very suitable compile target for high level language. Stack-base machine is very common in virtual machines design.

2 Stackless Bytecode

This article mentioned before Move compile to Move bytecode and run by MoveVM. However the Move bytecode is tightly integrated with Move bytecode verifier and MoveVM runtime, Every Move code need be verified before run and there is a lot of safety guarantee that hard to port to wasm.

Stackless Bytecode is refreshing simple bytecode than the MoveVM bytecode.

```
pub enum Bytecode {
    Assign(AttrId, TempIndex, TempIndex, AssignKind),

    Call(
        AttrId,
        Vec<TempIndex>,
        Operation,
        Vec<TempIndex>,
        Option<AbortAction>,
    ),
    Ret(AttrId, Vec<TempIndex>),

    Load(AttrId, TempIndex, Constant),
    Branch(AttrId, Label, Label, TempIndex),
    Jump(AttrId, Label),
    Label(AttrId, Label),
    Abort(AttrId, TempIndex),
    Nop(AttrId),

    SaveMem(AttrId, MemoryLabel, QualifiedInstId<StructId>),
    SaveSpecVar(AttrId, MemoryLabel, QualifiedInstId<SpecVarId>),
    Prop(AttrId, PropKind, Exp),
}
```

The TempIndex is index of local variables,local variable can mapping to wasm locals directly.

The Stackless Bytecode is register-based, When I see this IR,I know how to translate this IR to wasm directly.

For example of Assign.

```
Assign(AttrId, TempIndex, TempIndex, AssignKind),
// can generate list of wasm codes.
local.get right_local_index  // push right value on stack
local.set left_local_index   // store into local variable slots
```

A lot of Move functionality encode in *Call* bytecode.

```
Call(
    AttrId,
    Vec<TempIndex>,
    Operation,  //
    Vec<TempIndex>,
    Option<AbortAction>,
),

/// An operation -- target of a call. This contains user functions, builtin functions, and
/// operators.
#[derive(Debug, Clone, PartialEq, Eq)]
pub enum Operation {
    // User function
    Function(ModuleId, FunId, Vec<Type>),

    // Markers for beginning and end of transformed
    // opaque function calls (the function call is replaced
    // by assumes/asserts/gotos, but it is necessary to
    // add more assumes/asserts later in the pipeline.
    OpaqueCallBegin(ModuleId, FunId, Vec<Type>),
    OpaqueCallEnd(ModuleId, FunId, Vec<Type>),

    // Pack/Unpack
    Pack(ModuleId, StructId, Vec<Type>),
    Unpack(ModuleId, StructId, Vec<Type>),

    // Resources
    MoveTo(ModuleId, StructId, Vec<Type>),
    MoveFrom(ModuleId, StructId, Vec<Type>),
    Exists(ModuleId, StructId, Vec<Type>),

    // Borrow
    BorrowLoc,
    BorrowField(ModuleId, StructId, Vec<Type>, usize),
    BorrowGlobal(ModuleId, StructId, Vec<Type>),

    // Get
    GetField(ModuleId, StructId, Vec<Type>, usize),
    GetGlobal(ModuleId, StructId, Vec<Type>),

    // Builtins
    Uninit,
    Destroy,
    ReadRef,
    WriteRef,
    FreezeRef,
    Havoc(HavocKind),
    Stop,

    // Memory model
    IsParent(BorrowNode, BorrowEdge),
    WriteBack(BorrowNode, BorrowEdge),
    UnpackRef,
    PackRef,
    UnpackRefDeep,
```

```
    PackRefDeep,

    // Unary
    CastU8,
    CastU16,
    CastU32,
    CastU64,
    CastU128,
    Not,

    // Binary
    Add,
    Sub,
    Mul,
    ...

    // Debugging
    TraceLocal(TempIndex),
    ...
}
```

Porting *Call* bytecode typically involve next step.

1. loading value to stack.
2. perform operation.
3. store back.

If we are load every operator from memory looks inefficient, But typically wasm runtime has an optimization pass called mem2reg that transforms load(store too) to SSA registers.

For examples of porting Add.

```
call add inputs[
    1 // local slots 1
    2 // local slots 2
    ]  to outputs 3  // local slots 3

// can translate to
local.get 1  // load from local slots 1
local.get 2  // load from local slots 2
i32.add      // perform add
local.set 3  // store back.
```

Stackless IR is more simple to translate to wasm.

3 Branch

We have lowered some simple instruction to wasm, But I has not talk about how to port branch instruction.
Porting branch instruction is hard because of wasm doesn't support *goto* , and Stackless IR is register-based that contains conditional and unconditional branch.

```
    Branch(AttrId, Label, Label, TempIndex),
    Jump(AttrId, Label),
```

To solve this issue we need be able to build 'loop','if' statement from Stackless bytecode.
To understand What I am talking about you better understand some concept related to compiler design.

- Basic block
- Control-flow graph
- Dominator Tree

In compiler there are two ways of iter all Basic Block.

- postorder

```
// image the code below compile to Stackless bytecode

let i = 0;  // these two will visit lastly
let max = 10;

while(i < max) {  // visit thirdly
    // do something
    // visit secondly
}
// visit firstly
return
```

- preorder *just opposite of postorder*

successors,predecessors
The blocks to which control may transfer after reaching the end of a block are called that block's successors, while the blocks from which control may have come when entering a block are called that block's predecessors. The start of a basic block may be jumped to from more than one location.

3.1 Loop analyze

Let's use talk about loop analyze briefly.
Loop Header

```
fun main() {
    let i = 0;
    loop { // This is a loop header.
        i = i + 1;
        ...
    }
}
```

But I do we detect a loop header?
Base on some theory we have learn about compiler,detect loop header contains next steps.

1. We travel basic blocks in preorder.
2. If a block *dominates* it's any of *predecessors* then It is a loop header.

After we detect the loop header,we can find the loop depth,block belong to which loop,etc.

3.2 solve branch

Based on loop analyze we talked about before. We can solve the *branch target* by list of rules.

- target is loop header, this is a *continue* statement in Move.
- target have different loop depth(compare to current), This is *break* in Move.
- ... then This is a if statement.
- ...

4 Generation

So far We have analyzed the 'loop' and the 'if',generate code is relatively simple.

We just just generate basic block in preorder one by one, When we met a loop header, we generate a 'loop',etc.

5 Type Representation

We will need value's type in various places.

- vector push back
- vector drop (We need know vector's type to perform proper drop to avoid memory leak.)
- etc...

Value's type may represent as a Enum.

```
pub enum Type {
    Primitive(Primitive),
    ...
}
```

In compile Time we can save all Move value's type in some global storage,and pass it as argument to natives functions,etc.

6 Natives

6.1 Vector

6.1.1 Implementation

```
pub struct MoveVector {
    /// Safety: must be correctly aligned per type
    pub ptr: *mut u8,
    /// in typed elements, not u8
    pub capacity: u64,
    /// in typed elements, not u8
    pub length: u64,
    pub ety_length: u64, // cached,also can compute from ‘ety‘
    pub ety: &'static Type,
}
```

This is a code snippet of Rust, Vector can be implement in Rust, and Move make call the these code. We can go though a typically function implementation 'push back'.
Prototype:

```
#[bytecode_instruction]
/// Add element ‘e‘ to the end of the vector ‘v‘.
native public fun push_back<Element>(v: &mut vector<Element>, e: Element);
```

Implementation:

1. ensure *capacity* of the current vector because of capacity maybe not enough to accommodate the value.
2. Copy type length of data of e to v.

```
#[export_name = "move_native_vector_push_back"]
pub unsafe fn push_back(v: &mut MoveVector, e: *const AnyValue) {
    v.ensure_cap(v.length + 1);
    std::ptr::copy::<u8>(
        e,
        unsafe { v.ptr.add((v.ety_length * v.length) as usize) },
        v.ety_length as usize,
    );
    v.length += 1;
}
```

6.1.2 Drop

Vector is variable-length type, We can't know the size of the vector at compile time. So we must alloc vector on heap instead of stack.
MoveVM using Rust *Vec* ,*Rc* to implement a Move *vector* , So there is no code in MoveVM explicitly to drop the vector. But porting to wasm , When *return* We need to drop all the vector type in locals except values in return values.

6.1.3 Alloc vector on stack

Vector usually need alloc on heap,But if the vector never escape and vector's length is fixed,we can alloc on stack.

6.2 debug::print

Prototype like this.

```
fn move_debug_print(x: &AnyValue, t: &'static Type) {
    unimplemented!()
}
```

7 Thanks

Thanks for read this document, the document right now is very incomprehensive , I just outline most important things in my mind.

8 Useful links

- Move
- WebAssembly
- Wasmtime
- Wasm-tools
- Stackless Bytecode