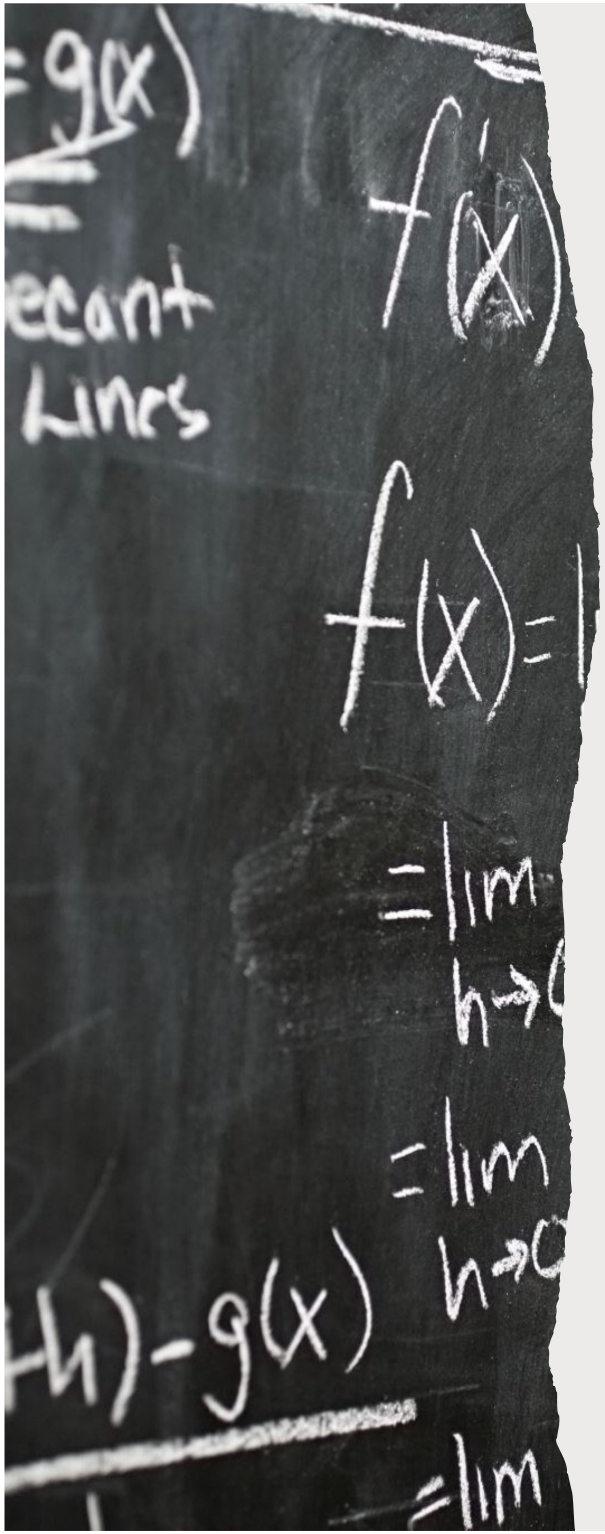


Symbolic Semantics and Intermediate Representation

The issue

- Compilers and analyzers work on Intermediate Representations (IRs) rather than source code.
- IRs are simplified, structured languages used for formal reasoning and program analysis.
- Examples: LLVM IR, Java bytecode, WebAssembly (WASM), … .
- Goal: Understand IR semantics and challenges of symbolic operational semantics.

Intermediate Representation

An IR is a machine-independent, typed language that exposes control and data flow explicitly.

Features: explicit control flow, explicit memory ops, static typing, close to machine level.

WASM (fragment)

```
(func $add
  (param $x i32)
  (param $y i32)
  (result i32)
  local.get $x
  local.get $y
  i32.add
)
```

WASM (fragment)

```
(func $add
  (param $x i32)
  (param $y i32)
  (result i32)
  local.get $x
  local.get $y
  i32.add
)
```

WASM is a stack machine

`(func $add … )` defines a function named `$add`.
`$add` takes two parameters, `$x` and `$y`, both 32-bit integers (`i32`).
`$add` returns a single 32-bit integer (`(result i32)`).

`$add`.body:
`local.get $x` pushes the value of `$x` onto the stack.
`local.get $y` pushes `$y`.
`i32.add` pops both values, adds them, and pushes the result (which becomes the return value).

The WASM Execution Model

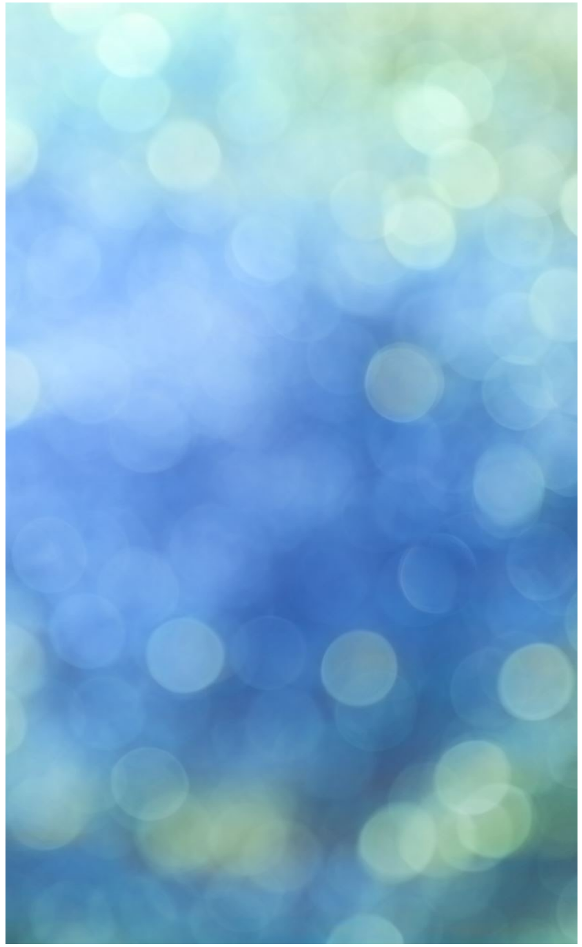
Stack-based virtual machine with no registers.

Linear memory: array of bytes private to the module.

Structured control: block, loop, if, br, br_if.

Sandboxed: cannot access host memory or syscalls.

Execution state: ⟨instr_seq, store, memory, locals, stack⟩.



Operational Semantics (Concrete)

Defines small-step transitions between states

$$\langle i32.const\ n, \sigma \rangle \rightarrow \langle \sigma \cdot push(n) \rangle$$

$$\langle i32.add\ \sigma \cdot n_1 \cdot n_2 \rangle \rightarrow \langle \sigma \cdot (n_1 + n_2 \rangle$$

$$\frac{if\ 0\ \leq a \leq mem.size}{\langle store\ a\ v, mem \rangle \rightarrow \langle mem[a = v] \rangle}$$

A core IR

An Intermediate programming language (syntax)

program	::=	*stmt**
stmt s	::=	*var* := *exp* $\mid$ store(*exp*, *exp*)
		$\mid$ goto *exp* $\mid$ assert *exp*
		$\mid$ if *exp* then goto *exp*
		else goto *exp*
exp e	::=	load(*exp*) $\mid$ *exp* $\Diamond_b$ *exp* $\mid$ $\Diamond_u$ *exp*
		$\mid$ *var* $\mid$ get_input(src) $\mid$ v
$\Diamond_b$	::=	typical binary operators
$\Diamond_u$	::=	typical unary operators
value v	::=	32-bit unsigned integer

Remark

The expression ***get_input(src)*** returns input from the source stream ***src***.

We model input stream as a suitable list,

scr = v:: src’

We omit the type-checking mechanism of our language and assume things are well-typed in the obvious way,

Run-time structures

- Σ: the ordered sequence of program statements
  Σ = Nat -> Stmt
- μ: memory **μ: Loc -> Values**
- ρ: environment **ρ: Var -> Loc + Values**
- **pc**: program counter
- ι: next instruction

Program evolution: expressions

$$\mu, \rho \ \vdash e \Downarrow v$$

Intuition: evaluationg the expression e in the run-time context provided by the memory μ and the environment ρ produces v as result

Program evolution: statements

$$\Sigma, \mu, \rho, pc\colon\ smt \rightarrow \Sigma, \mu', \rho', pc' \colon smt'$$

- **Intuition: the execution of the statement smt in the run-time context given by**
  - **the program list (Σ),**
  - **the currentr memory state (μ),**
  - **the current binding for variable (ρ)**
  - **the current program counter (pc)**
- **yields a new state of program execution (Σ, μ', ρ', pc')**

$$\Sigma, \mu, \rho, pc\colon\ smt \rightarrow \Sigma, \mu', \rho', pc'\colon smt'$$

Remark

- **Intuition: the execution of the statement smt in the run-time context yields a new state of program execution (Σ, μ´, ρ´, pc´)**
- **The program Σ does is not modified by transitions.**
  - **We do not allow programs with dynamically generated code.**

A sample of the operational semantics (expressions)

$$\frac{src = v :: src'}{\mu, \rho \vdash getInput(src) \Downarrow v}$$

$$\frac{\mu, \rho \vdash e \Downarrow v_1 \quad v = \mu(v_1)}{\mu, \rho \vdash load\ e \Downarrow v}$$

$$\frac{}{\mu. \rho \vdash var \Downarrow \rho(var)}$$

A sample of the operational semantics (statement)

$$\frac{\mu, \rho \vdash e \Downarrow v \qquad \rho' = \rho[var = v] \quad \iota = \Sigma[pc + 1]}{\Sigma, \mu, \rho, pc\colon\ var = e\ \rightarrow \Sigma, \mu,\ \rho', pc + 1\colon\ \iota}$$

A sample of the operational semantics (statement)

$$\frac{\mu, \rho \vdash e \Downarrow v \qquad \rho' = \rho[var = v] \quad \iota = \Sigma[pc + 1]}{\Sigma, \mu, \rho, pc\colon\ var = e \ \rightarrow \Sigma, \mu,\ \ \rho', pc + 1\colon\ \iota}$$

The current state of execution

A sample of the operational semantics (statement)

Evaluation of the expression

$$\frac{\mu, \rho \vdash e \Downarrow v \qquad \rho' = \rho[var = v] \quad \iota = \Sigma[pc + 1]}{\Sigma, \mu, \rho, pc\colon\ var = e \ \rightarrow \Sigma, \mu,\ \ \rho', pc + 1\colon\ \iota}$$

The current state of execution

A sample of the operational semantics (statement)

The new binding

Evaluation of the expression

The continuation (next instruction)

$$\frac{\mu, \rho \vdash e \Downarrow v \qquad \rho' = \rho[var = v] \quad \iota = \Sigma[pc + 1]}{\Sigma, \mu, \rho, pc\colon\ var = e \ \rightarrow \Sigma, \mu, \ \rho', pc + 1\colon\ \iota}$$

The current state of execution

A sample of the operational semantics (statement)

The new binding

Evaluation of the expression

The continuation (next instruction)

$$\frac{\mu, \rho \vdash e \Downarrow v \qquad \rho' = \rho[var = v] \quad \iota = \Sigma[pc + 1]}{\Sigma, \mu, \rho, pc\colon\ var = e \ \rightarrow \Sigma, \mu, \ \ \rho', pc + 1\colon\ \iota}$$

The current state of execution

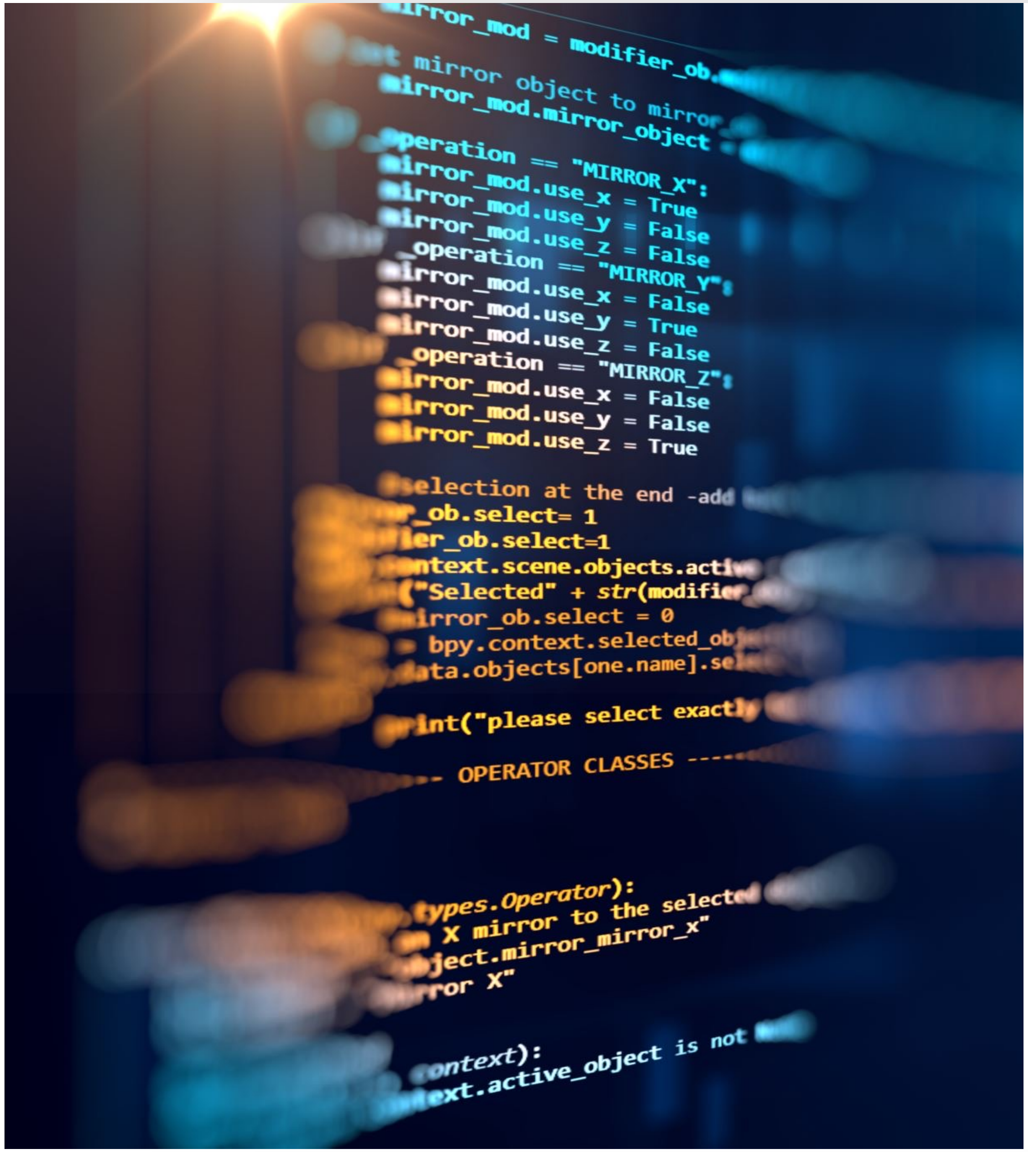
The next state of execution

A sample of the operational semantics (statements)

$$\frac{\mu, \rho \vdash e \Downarrow v_1 \quad \iota = \Sigma[v_1]}{\Sigma, \mu, \rho, pc: goto\ e \rightarrow \Sigma, \mu, \rho, v_1: \iota}$$

$$\frac{\mu, \rho \vdash e_1 \Downarrow v_1 \quad \mu, \rho \vdash e_2 \Downarrow v_2 \ \iota = \Sigma[pc + 1] \ \ \mu' = \mu[v_1 = v_2]}{\Sigma, \mu, \rho, pc: Store(e_1, e_2) \rightarrow \Sigma, \mu', \rho, pc + 1: \iota}$$

$$\frac{\mu, \rho \vdash e \Downarrow 1 \quad \iota = \Sigma[pc + 1]}{\Sigma, \mu, \rho, pc: assert(e) \rightarrow \Sigma, \mu, \rho, pc + 1: \iota}$$



What about functions?

Function calls in high-level programming language are compiled by storing the return address and transferring control flow.

From Concrete to Symbolic Semantics

- Symbolic execution replaces concrete values by symbolic variables.
- Path conditions (Π) represent branch decisions as logical formulas.

Symbolic Operational Semantics

Symbolic state

$$\Pi, \Sigma, \mu, \rho, pc: \; smt$$

- **the program list (Σ),**
- **the currentr memory state (μ),**
- **the current binding for variable (ρ)**
- **the current program counter (pc)**
- → **The current path condition (Π)**

S-INPUT

$$\frac{v\ fresh\ symbolic\ constant}{\mu, \rho \vdash getInput() \quad \Downarrow v}$$

S-ASSERT

$$\frac{\mu, \rho \vdash e \downarrow e' \; \Pi' = \Pi \;\wedge (e' = true) \quad \iota = \Sigma[pc + 1])}{\Pi, \Sigma, \mu, \rho, pc: assert(e) \rightarrow \Pi', \Sigma, \mu, \rho, pc + 1: \iota}$$

S-COND-TRUE

$$\frac{\mu, \rho \vdash e \downarrow e' \quad \mu, \rho \vdash e_1 \downarrow v_1 \quad \Pi' = \Pi \wedge (e' = 1) \quad \iota = \Sigma[v_1]}{\Pi, \Sigma, \mu, \rho, pc\text{: } if\ \ e\ then\ goto\ e_1 else\ goto\ e_2 \rightarrow \Pi', \Sigma, \mu, \rho, v_1\text{: } \iota}$$

S-COND-TRUE

$$\frac{\mu, \rho \vdash e \downarrow e' \quad \mu, \rho \vdash e_1 \downarrow v_1 \quad \Pi' = \Pi \wedge (e' = 1) \quad \iota = \Sigma[v_1]}{\Pi, \Sigma, \mu, \rho, pc: if\ e\ then\ goto\ e_1 else\ goto\ e_2 \rightarrow \Pi', \Sigma, \mu, \rho, v_1: \iota}$$

STRONG ASSUMPTION: v_1 must be an actual value

S-COND-TRUE

$$\frac{\mu, \rho \vdash e \downarrow e' \quad \mu, \rho \vdash e_1 \downarrow v_1 \quad \Pi' = \Pi \wedge (e' = 1) \quad \iota = \Sigma[v_1]}{\Pi, \Sigma, \mu, \rho, pc: if\ e\ then\ goto\ e_1 else\ goto\ e_2 \to \Pi', \Sigma, \mu, \rho, v_1: \iota}$$

STRONG ASSUMPTION: v_1 must be an actual value

How can we encode this constraint?

Statement	ρ	Π	pc
start	{}	true	1
x= 2*get_input()	{x = 2 *s}	true	2
If x-5 =14 then goto 3 else goto 4	{x = 2 *s}	((2*s) – 5 = 14)	3
If x-5 =14 then goto 3 else goto 4	{x = 2 *s}	!((2*s) – 5 = 14)	4

Challenging issue

- What should we do when the analysis uses the memory model (μ) whose index must be a non-negative integer with a *symbolic index?*

Memory model

- Memory is a **linear memory:** a contiguous byte array.
  - It starts at **0** and grows in multiples of 64 KB “pages”.
- Addresses are **integers** (no pointers or segmentation).
- There are **no raw pointers to the host**.
- Memory operations are **explicit**:

Memory model (semantically)

- The memory is a function

$$\mu: Addr \to Byte$$

Symbolic View of Linear Memory

- In symbolic execution, the memory is modeled as a **symbolic array**, typically using the SMT **theory of arrays**:

```
Mem:Array(Int, Byte)
```

Mem Op	Symbolic Encoding
store a v	mem' = store(mem, a, v)
load a	x = select(mem, a..a+3) (4 bytes)

Memory Ops

Mem Op	Symbolic Encoding
store a v	mem' = store(mem, a, v)
load a	x = select(mem, a) (4 bytes)

Memory Ops

mem’ is the same as mem except at index a, where the value is v.
Subsequent reads are performed on mem’.

Mem Op	Symbolic Encoding
store a v	Mem’ = store(mem, a, v)
load a	x = select(mem, a) (4 bytes)

Memory Ops

Symbolic Addresses

Concrete addresses

If the address is concrete (e.g. `a = 100`), easy:

```
mem' = store(Mem, 100, v).
```

Symbolic addresses

If `a` is symbolic, e.g. `a =` a_0, the read or write cannot be resolved concretely.

Symbolic Addresses

Symbolic addresses

Reads produce a conditional expression:

$$select(store(mem, a_1, v_1), a_2) =$$
$$ite(a_2 = a_1, v_1, select(mem, a_2))$$

Writes create a new memory expression with that conditional update.

This leads to **nested ITE chains** or **deep store/select terms**, which can grow quickly and stress SMT solvers.

Memory: grow and bound

The symbolic executor must:

1. Maintain a symbolic bound `MEM_SIZE`.
2. Add guards on every access:

$$0 \leq a < MEM\_SIZE$$

When memory grows, update:

$$MEM\_SIZE' = MEM\_SIZE + 64KB * n$$

)where `n` may be symbolic or concrete).

Address Space Partitioning (WASM)

WASM forbids overlapping segments unless the program explicitly overwrites memory.

WASM Executors partition memory into **disjoint regions** (stack, heap, globals).

Symbolically: each region is a separate symbolic array:

`StackMem, HeapMem, GlobalMem`

This avoids having a single enormous symbolic array term for the entire 4 GB space.

Challenges: Memory model

- Symbolic memory: nested ITEs and large SMT terms.
- Path explosion: every conditional doubles states.
- Bounds: must encode $0 \le addr < MEM\_SIZE$.

Symbolic Load/Store (WASM)

```
(local.get $addr)
(local.get $val)
(i32.store)

(local.get $addr)
(i32.load)
```

`local.get $addr` pushes the address onto the stack.
`local.get $val` pushes the value.
`i32.store` pops both (address and value) and writes the value into linear memory at the given address.

The second pair of statements (`local.get $addr`, `i32.load`) reads back (`load`) the 32-bit value from the same memory address.

Symbolic Load/Store (WASM)

- Symbolic SMT model:

$$mem' = store(mem, a_0, v_0); select(mem' a_0)$$

Intuition: $x = v_0$

Symbolic Load/Store (WASM)

- Symbolic SMT model:

$$mem' = store(mem, a_0, v_0); select(mem' a_0)$$

Intuition: $x = v_0$

- But this is not enough: Add bounds constraint:

$$0 \le a_0 \le MEM\_SIZE$$

Overall

```
(memory 1)
(func $f (param $addr i32) (param $v i32)
(result i32)
  local.get $addr
  local.get $v
  i32.store       ;; Mem1 = store(Mem0, addr, v)
  local.get $addr
  i32.load        ;; result = select(Mem1, addr)
)
```

```
(memory 1)
(func $f (param $addr i32) (param $v i32) (result i32)
  local.get $addr
  local.get $v
  i32.store       ;; Mem1 = store(Mem0, addr, v)
  local.get $addr
  i32.load        ;; result = select(Mem1, addr)
)
```

x = v

```
; Variables
(declare-fun Mem0 () (Array (_ BitVec 32) (_ BitVec 8)))
(declare-fun a () (_ BitVec 32))
(declare-fun v () (_ BitVec 32))
; 4-byte store and load
(define-fun Mem1 () (Array (_ BitVec 32) (_ BitVec 8))
  (store (store (store (store Mem0 a ((_ extract 7 0) v))
                  (bvadd a #x00000001) ((_ extract 15 8) v))
                  (bvadd a #x00000002) ((_ extract 23 16) v))
                  (bvadd a #x00000003) ((_ extract 31 24) v)))
(define-fun x () (_ BitVec 32)
  (concat (select Mem1 (bvadd a #x00000003))
        (select Mem1 (bvadd a #x00000002))
        (select Mem1 (bvadd a #x00000001))
        (select Mem1 a)))
```

```
(memory 0)
(func $f (param $addr i32) (param $v i32) (result i32)
  local.get $addr
  local.get $v
  i32.store       ;; Mem1 = store(Mem0, addr, v)
  local.get $addr
  i32.load        ;; result = select(Mem1, addr)
)
```

```
(declare-fun Mem0 () (Array (_ BitVec 32) (_ BitVec 8)))
(declare-fun a () (_ BitVec 32))
(declare-fun v () (_ BitVec 32))
```

`Mem0` is the linear memory modeled as an **array** from 32-bit addresses to 8-bit bytes:
`Mem0 : (Array BV32 → BV8).`

`a` is a 32-bit **address** (bit-vector).
`v` is a 32-bit **word**

The SMT **array theory** is the standard way to model memory and loads/stores.

```
(memory 0)
(func $f (param $addr i32) (param $v i32) (result i32)
  local.get $addr
  local.get $v
  i32.store       ;; Mem1 = store(Mem0, addr, v)
  local.get $addr
  i32.load        ;; result = select(Mem1, addr)
)
```

```
(define-fun Mem1 () (Array (_ BitVec 32) (_ BitVec 8))
  (store (store (store (store Mem0 a ((_ extract 7 0) v))
                (bvadd a #x00000001) ((_ extract 15 8) v))
                (bvadd a #x00000002) ((_ extract 23 16) v))
                (bvadd a #x00000003) ((_ extract 31 24) v)))
```

`store` writes 4 bytes of `v` into memory starting at `a` :

- Byte 0 (least significant 8 bits) at address `a`
- Byte 1 at `a + 1`
- Byte 2 at `a + 2`
- Byte 3 (most significant 8 bits) at `a + 3`

`(_ extract 7 0) v` takes the **lowest** 8 bits of `v` .
`(_ extract 15 8) v` is the next 8 bits, and so on.
`bvadd` does 32-bit modular addition on addresses.

each `(store M i b)` returns a **new array** that maps address `i` to byte `b` and leaves all other addresses as in `M`. Nesting the four `store`s yields `Mem1`, which differs from `Mem0` **only** at `a, a+1, a+2, a+3`.

```
(memory 0)
(func $f (param $addr i32) (param $v i32) (result i32)
  local.get $addr
  local.get $v
  i32.store       ;; Mem1 = store(Mem0, addr, v)
  local.get $addr
  i32.load        ;; result = select(Mem1, addr)
)
```

```
(define-fun Mem1 () (Array (_ BitVec 32) (_ BitVec 8))
  (store (store (store (store Mem0 a ((_ extract 7 0) v))
                 (bvadd a #x00000001) ((_ extract 15 8) v))
                 (bvadd a #x00000002) ((_ extract 23 16) v))
                 (bvadd a #x00000003) ((_ extract 31 24) v)))
```

```
Mem1[a] = v[7:0]
Mem1[a+1] = v[15:8]
Mem1[a+2] = v[23:16]
Mem1[a+3] = v[31:24]
```

All other addresses equal `Mem0`.

```
(memory 0)
(func $f (param $addr i32) (param $v i32) (result i32)
  local.get $addr
  local.get $v
  i32.store       ;; Mem1 = store(Mem0, addr, v)
  local.get $addr
  i32.load        ;; result = select(Mem1, addr)
)
```

```
(define-fun x () (_ BitVec 32)
  (concat (select Mem1 (bvadd a #x00000003))
          (select Mem1 (bvadd a #x00000002))
          (select Mem1 (bvadd a #x00000001))
          (select Mem1 a)))
```

`load` reads 4 bytes starting at `a` and reassembles a 32-bit word
`(select Mem1 i)` reads the byte at address `i`.
`concat` packs 4 bytes into 32 bits in order of the **word**:
the leftmost argument becomes the **most significant** byte of `x`.
Using the selects in the order `(a+3, a+2, a+1, a)` exactly reconstructs the 32-bit value.

```
(memory 0)
(func $f (param $addr i32) (param $v i32) (result i32)
  local.get $addr
  local.get $v
  i32.store        ;; Mem1 = store(Mem0, addr, v)
  local.get $addr
  i32.load         ;; result = select(Mem1, addr)
)
```

```
(define-fun x () (_ BitVec 32)
  (concat (select Mem1 (bvadd a #x00000003))
          (select Mem1 (bvadd a #x00000002))
          (select Mem1 (bvadd a #x00000001))
          (select Mem1 a)))
```

he selects return:

```
select Mem1 a = v[7:0]
select Mem1 a+1 = v[15:8]
select Mem1 a+2 = v[23:16]
select Mem1 a+3 = v[31:24]
```

x = concat(v[31:24], v[23:16], v[15:8], v[7:0]) = v

Why this is correct (array axioms)

The theory-of-arrays gives two key equalities:

1.Read-after-write (same index):

`select(store(M, i, v), i) = v`

2.Read-after-write (different index):

`i ≠ j ⇒ select(store(M, i, v), j) = select(M, j)`

Applying these four times to the nested stores yields exactly the bytes we expect at `a, a+1, a+2, a+3`. Concatenating them recreates `v`.

Summary & Takeaways

- IRs like WASM enable precise formal reasoning.
- Concrete semantics: deterministic transitions.
- Symbolic semantics: generalize to formulas over symbols.
- Main hurdles: symbolic memory, symbolic adresses.
- Solutions: (SMT) abstract domains, invariants, regioned memory, concolic testing.