Knowledge Usage Description

Instruction Format. We require the LLM to represent the format of an instruction as a mapping from the positions of its operands and immediate arguments to their corresponding types. This knowledge is critical for Code Insertion Mutation and Target Instruction Insertion Mutation since it helps to represent the instruction's operand and immediate arguments.

Instruction Type. We query the LLM for the type representation of an instruction, which is expressed as a list of candidate types. Each candidate corresponds to a specific function type or expression. For instance, Figure 1 illustrates the type representation of the `local.set` instruction, showing that `local.set` takes an operand matching the type of the local variable at index `imm_0` and produces no output. This knowledge is critical for inferring the types of instructions, enabling the generation of valid instruction sequences. It plays an essential role in both Code Insertion Mutation and Target Instruction Insertion Mutation, as it ensures type consistency in the generated Wasm code.

```
[
  {
    "param": ["context.locals[imm_0]"],
    "result": []
  }
]
```

Figure 1: The type representation of `local.set`.

Validation Rules. Validation rules define the constraints an instruction must satisfy to be considered valid. An instruction is valid only if all the specified constraints are satisfied. We use the prompt to guide the LLM to represent these validation rules as either type constraints or numeric constraints.

Type constraints specify the validation rules for the types of operands. In the LLM's response, they are represented as a mapping with three fields: the two type representations (e.g., `i32`, *op_0.type*) and the relationship between them (e.g., equality, inclusion, or their negation). For example, Figure 2 illustrates the type constraints for the `i32.store` instruction, which requires two operands of type '$i32$.

We use numeric constraints to represent the other validation rules. For instance, Figure 3 shows a validation rule generated by the LLM, stating that the number of memories must be greater than zero. This rule can be expressed as the numeric constraint: context.mems.length $>$ 0. It indicates that if there is no memory defined in the test case, the `i32.store` instruction is deemed invalid.

For the Code Insertion Mutation, validation rules are crucial for generating valid instructions by guiding both instruction selection and the generation of immediate arguments. For the Target

```
[
  {
    "v1": "op_0.type",
    "v2": "i32",
    "relation": "eq"
  },
  {
    "v1": "op_1.type",
    "v2": "i32",
    "relation": "eq"
  }
]
```

Figure 2: ype constraints of `i32.store`, specifying that it takes two operands of type $i32$.

```
{
    "v1": "context.mems.length",
    "v2": "0",
    "relation": "gt"
}
```

Figure 3: Numeric constraint of 'i32.store', indicating that the number of memories must be greater than zero.

Instruction Insertion Mutation, these rules are used to create instructions that either conform to the validation rules or deliberately violate a specific rule. This allows us to verify whether the validation logic of an instruction is implemented correctly.

Execution Behavior Conditions. Execution behavior conditions specify the conditions under which an instruction produces specific outcomes. We use the prompt to guide the LLM in representing these conditions as numeric constraints, as illustrated in Figure 3. For example, for the `i32.store` instruction, the condition for successfully storing a byte is represented as imm_1 + op_1 + 4 < context.Mem[0].length, where imm_1 and op_1 represent the immediate argument and operand that determine the storage location.

This information plays a critical role in guiding Target Instruction Insertion Mutation to create test instructions with precise immediate arguments and operands that demonstrate representative behaviors. Specifically, for each execution behavior condition C, we solve a combination of the numeric constraints derived from validation rules and C to determine the immediate arguments and operands necessary for the instruction to exhibit the targeted behavior.

Control Flow Constructs. We ask the LLM to summarize the control flow constructs as a grammar, which helps Code Insertion Mutation to generate to generate Wasm code with complex control

flow.

Module Definition Format. The module definition format is represented as a mapping from the names of the fields in the definition to their types. The knowledge guides Module Definition Mutation to generate valid module definitions by filling each field with the data of the valid type.