Documentation for Slang bytecode, the processing level code used for the Slang programming language created by Jonathan Huo and Ethan White in 2016.

Whitespace is not significant in SKB, with the exception of values in PLISTS.

Each top-level construct in SKB is composed of smaller parts. One of those parts is known as a *PLIST*, which is simply a container for a set of key-value pairs. It is begun with an opening curly brace token, followed by some lines of text, followed by a closing curly brace on its own line. Each line of text is divided into two parts: the first word, and the rest of the content. The first word is used as the key, whereas the rest of the content is used as the value. The following is an example of a PLIST.

```
{
   key1 value1
   key2 Key two has a more complex value.
   key3 The spaces are included literally in the value.
}
```

Another one if these parts is an *instruction listing*, composed of an opening curly brace, followed by an ordered set of *instruction declarations*, followed by a closing curly brace. An instruction declaration is composed of an identifier, followed by any number of tokens, either numbers or identifiers, delimited by commas, followed by a semicolon. The following is an example of an instruction listing.

```
{
    ldint32 2, 0;
    callret 0, double, 0;
    goto 1;
}
```

There are two top-level objects in SKB: *structs* and *methods*. A struct is defined by the token “struct”, followed by two PLISTs; the first PLIST is for general properties about the struct, whereas the second associates property names with types (initial types have no representation at the bytecode level). A method is defined by a the token “method”, followed by a PLIST, followed by an instruction listing. The following is an example of a full SKB program.

```
struct {
   name NumberContainer
} {
```

```
   int32 the_number
}
method {
   ctor NumberContainer
   maxregisters 2
   params int32
} {
   ldthis 1;
   structpropset 0, the_number, 1;
   return;
}
method  {
   name main
   maxregisters 2
} {
   ldint32 5, 0;
   structctor 0, NumberContainer, 1;
   ldstructprop 0, the_number, 1;
   return 1;
}
```

The Semantics of SKB - A.2

PLISTS describe basic properties of bytecode objects, such as their name. SKB execution begins at the start of the main method, and continues based on an instruction pointer that is incremented by an amount specified in the instruction definition.

The first identifier in an instruction is known as the *opcode*, with all other tokens that are neither commas nor periods being parameters. Structs can be viewed as maps from keys (property names) to values (of the type defined in the struct definition). There exist instructions to operate on structs.

The PLIST of a method may indicate that, for example, it is the constructor of a struct.

PLIST Pairs - A.3

PLIST Pair :: `returns`

Specifies the return type. Must be a type.

PLIST Pair :: `args`

Specifies the types of the arguments that the method takes. This must be a space-separated list. The nth argument will be put into register n.

PLIST Pair :: `maxregisters`

Specifies the maximum number of registers needed during the execution of this method. If this is ever wrong, it will cause a VM segfault, and *may allow remote code execution in a network application*. This includes the registers used for the storage of arguments.

PLIST Pair :: `impl`

Asserts that this method is an implementation for the struct with the name equal to the value of impl. This is optional.

PLIST Pair :: `operator`

Asserts that this method is the operator specified in the value of this declaration. This is optional; however, the method that declares this must be an impl of a struct. (This is not currently implemented).

PLIST Pair :: `constructor`

Specifies that this method is the constructor for the struct that this struct is an impl of.

Opcodes - A.4

Opcode :: (add, subtract, multiply, divide, mod, shr, shl)(uint64, int64, uint32, int32, uint16, int16, uint8, int8, f64, f32) reg1, reg2, out

(add, subtract, multiply, divide, modulus, shift right, shift left) register *reg1* and register *reg2* and put the output into register *out*.

Opcode :: (xor, or, and) reg1, reg2, out

(xor, or, and) register *reg1* and register *reg2* and put that into register *out*. Note that this is type-agnostic.

Opcode :: goto linenum

GOTO the line *linenum*. This is zero-based. In reality, this goes to the *linenum*th instruction, rather than the *linenum*th line.

Opcode :: (ifeq, ifneq) reg1 reg2 linenum

If the value in the register *reg1* is (equal, not equal) to the value in the register specified in the register *reg2*, GOTO the line number *linenum*.

Opcode :: (lt, gt, lteq, gteq)(uint64, int64, uint32, int32, uint16, int16, uint8, int8, f64, f32) reg1 reg2 reg3

If the the value in register *reg1* is (less than, greater than, less than or equal to, greater than or equal to) the value in register *reg2*, set the value in register *reg3* to 1. Otherwise, *do nothing*. *(Note: The reason I do nothing instead of setting the register to a 0 is for the branch predictor).*

Opcode :: ldthis reg1

Load the value of `this` (i.e. the struct that the currently executing method is a member of, possibly as a constructor) into *reg1*.

Opcode :: call arg0, arg1, ..., methodname

Call the method *methodname*, passing the parameters specified in registers *arg0*, *arg1*, *arg2*, etc. These will be put into registers 0, 1, 2, etc., respectively, of the resulting stack frame. Although it is technically legal not to pass any arguments, it is of limited utility, as the called methodname cannot generally affect any state (the exception being through filesystem access, network access, and the likes).

Opcode :: callret arg0, arg1, ..., methodname, return

Call the method *methodname*, passing the parameters specified in registers *arg0*, *arg1*, *arg2*, etc., and put the return value of the method into the register *return*. The parameters will be put into

registers 0, 1, 2, etc., respectively, of the resulting stack frame. The above arguments about effects on state do not apply.

Opcode :: structctor arg0, arg1, ..., structname, out

Instantiate the struct *structname* using the default constructor and the arguments in the registers *arg0*, *arg1*, *arg2*, etc., with the same method of passing parameters as above, and put the resultant struct in the register *out*.

Opcode :: structctornamed arg0, arg1, ..., structname, ctorname, out

Instantiate the struct *structname* using the constructor named *structname* and the arguments in the registers *arg0*, *arg1*, *arg2*, etc., with the same method of passing parameters as above, and put the resultant struct in the register *out*.

Opcode :: structcallret arg0, arg1, ..., methodname, structreg, return

Call the method *methodname* on the struct in register *structreg* with the arguments specified in the registers *arg0*, *arg1*, etc., and put the return value in the register *return*. Note that the type of the struct is inferred at runtime based on the type present upon the first invocation of the instruction.

Opcode :: structcall arg0, arg1, ..., methodname, structreg

Call the method *methodname* on the struct in register *structreg* with the arguments specified in the registers *arg0*, *arg1*, etc. Note that the type of the struct is inferred at runtime based on the type present upon the first invocation of the instruction.

Opcode :: ldstructprop structreg, propname, outreg

Load the property *propname* of the struct in the register *structreg* into the register *outreg*.

Opcode :: structpropset structreg, propname, inreg

Get the value in register *inreg* and put it into the struct in the register *structreg* under the property name *propname*.

Opcode :: arrget arrreg, indexreg, out

Read the value of the register *indexreg* as interpret it as an integer; call that *i*. Take value of the array in the register *arrreg* at the *i*th index and put it into the register *out*.

Opcode :: arrset arreg, indexreg, in

Read the value of the register *indexreg* as interpret it as an integer; call that *i*. Set value of the array in the register *arrreg* at the *i*th index to the value in the register *in*.

Opcode :: arralloc arrreg, lenreg

Allocates an array with length specified in the register *lenreg* into the register specified by *arrreg*.

Opcode :: return [reg]

If *reg* is specified, get the value in register *reg*, and call that *v*. Then, pop a stack frame, restoring the state specified in the previous stack frame. Then, if *reg* as specified, put the value *v* into the return register specified by the previous method. If there was no return value specified by the previous method, then the behaviour is left undefined, *although this could potentially allow remote code execution*.