The "Mobius" Slate Implementation Manual

Brian T. Rice and Lee Salzman

8th August 2004

Abstract

We introduce the various implementation mechanisms, structure, and their usage within the Slate system. This also gathers all notes about the implementation details, so that users of the system have a relatively easy entry into developing their own extensions or uses.

Contents

1 Overview **3**

2 Front-End **3**
- 2.1 Lexer . . . 3
- 2.2 Parser . . . 3
- 2.3 Code-walking . . . 3
- 2.4 Macros . . . 3
- 2.5 Modes . . . 3

3 Memory Usage and Layout **3**
- 3.1 Memory Structure Formats . . . 3
  - 3.1.1 Pointers . . . 3
  - 3.1.2 Headers . . . 4
  - 3.1.3 Objects . . . 4
  - 3.1.4 Arrays . . . 4
  - 3.1.5 Maps . . . 5
  - 3.1.6 Methods . . . 6
- 3.2 Core Resident Structures . . . 6
- 3.3 Memory-Management . . . 6
  - 3.3.1 Generational Mode . . . 6
  - 3.3.2 Non-moving Mode . . . 6
- 3.4 Interaction Structures . . . 6
  - 3.4.1 External Resource Handles . . . 6
  - 3.4.2 FFI . . . 6

4 C Translator and Dialect **6**
4.1 Overview . . . 6
4.2 Conventions . . . 7
4.3 Restrictions . . . 7
4.4 Special Methods . . . 8
4.4.1 Arithmetic / Logical Operators . . . 8
4.4.2 Conditionals . . . 8
4.4.3 Iteration . . . 8
4.4.4 Memory Access . . . 8

5 Bytecode Execution Engine **9**
5.1 Context Management . . . 9
5.1.1 Stack Format . . . 9
5.1.2 Establishment . . . 9
5.1.3 Disestablishment . . . 10
5.2 Primitives . . . 10
5.3 Native Methods . . . 10

6 Optimizing Compiler **12**
6.1 Intermediate Representation . . . 12
6.2 Machine Representation . . . 12
6.3 Optimizer . . . 12
6.4 Back-End . . . 12

List of Tables

1 Lexical Context Format . . . 9
2 Frame Format . . . 9
3 Block Format . . . 9
4 Normal Opcodes . . . 11
5 Extended Opcodes . . . 11

1 Overview

Slate's implementation is designed to provide an open and flexible means of extending and maintaining the system. The ability to run code optimally is an important factor in the design, but the overriding principles are to be portable and generic. Most importantly, the design tries to avoid making any design decisions too firmly assumed; that is, that the coupling between subsystems is loose, and particular design choices are (or should be) localized as much as possible.

2 Front-End

The implementation front-end works with source code from user input until it is converted into a form directly consumable for execution. All of these components are within the `Syntax` namespace.

Source-code parsing is performed by a two-stage system: incoming text from a stream is lexified into tokens, its external protocol being a stream of tokens. The parser is a further `ProcessorStream` of this token stream; users of the parser are provided with a stream of parse-node objects which represent the abstract syntax tree of Slate source. The compilers and other source tools work on the output of this stream.

2.1 Lexer

2.2 Parser

2.3 Code-walking

2.4 Macros

Macro-level method calls are evaluated and replaced in place with the call `macroExpand`, working recursively on whatever node is the argument.

2.5 Modes

3 Memory Usage and Layout

Part of the abstract machine design involves providing a layer of safe references and arithmetic, as well as defining the expected layout of targets of references.

3.1 Memory Structure Formats

3.1.1 Pointers

Object pointers are tagged with one bit, in the low end. A one value indicates a SmallInteger direct value stored in the remaining bits, a signed 31-bit (word-size

minus one) value with negatives in twos-complement format. A value of zero indicates a reference to an object on the heap.

3.1.2 Headers

Objects and other complex heap structures begin with a single-word header. The format:

Bit/Range	Interpretation
0	GC Mark Flag
1	Forwarding Flag
2...23	Identity-based Hash
24...29	Object Size
30...31	Format Code

The format codes are as follows:

Code	Interpretation
00	Normal Object
01	Array of ObjectPointers
10	Array of Byte Values
11	Payload - Mixed Slots and Array

"Payload" style objects contain an extra word, the low 30 bits of which are a size field denoting the number of slots. The high two bits are another format code for the payload itself.

3.1.3 Objects

The word format:

Word	Interpretation
0	Header
1	Map Pointer
2?	Optional Extended Object Size
2	Slot Value Pointer 0
2+N	Slot Value Pointer N

The slot values are stored in offsets specified by the object's map. The object format includes an optional extended size field before the slot values if the header's size field is maximized; that is, if its value is 255.

Delegate slots are stored in a contiguous block before non-delegating data slots.

3.1.4 Arrays

The word format:

Word	Interpretation
0	Header
1	Map Pointer
2?	Optional Extended Array Size
2	Element Value Pointer 0
2+N	Element Value Pointer N

This is adjusted for the word-size for the element type, say for a FloatArray or ByteArray. There is an optional extended size field just as for ordinary objects. The map pointer will refer to one of the VM-known array map objects.

3.1.5 Maps

The word format:

Word/Range	Interpretation
0	Header
1	Representative Object
2	Number of Delegation Slots
3	Number of Data Slots
4	Pointer to the Method Dependency Array
5	Oldest Generation of Weak References
6-7	Method Dispatch ID (a serial)
8	Visited Position Bitmask
9	Pointer to the Slot Entry Array
10	Pointer to the Role Entry Array

Slot Entries The slot entry word format:

Word	Interpretation
0	Slot Name Pointer
1	Slot Offset within the Object

The slot entry array word format (with optional extended size field):

Word	Interpretation
0	Header
1	Slot Entry 0
1 + 3 * N	Slot Entry N

Role Entries The role entry word format:

Word	Interpretation
0	Role Position Bitmask
1	Dispatch Position Bitmask
2	Method Pointer

The role entry array word format (with optional extended size field):

Word	Interpretation
0	Header
1	Role Entry 0
1 + 3 * N	Role Entry N

3.1.6 Methods

Word format additions (to... ?):

Word/Range	Interpretation
0-1	Dispatch ID
2	Dispatch Position Mask
3	Found Roles Position Bitmask
4	Dispatch Rank

3.2 Core Resident Structures

3.3 Memory-Management

3.3.1 Generational Mode (Not Fully Tested)

Slate has an incremental 2-generation garbage collector, using a mark-sweep-compact algorithm at its core, coded to avoid the need for a mark-stack. There is also a facility for relocating objects efficiently using forwarding blocks. General care has been taken to separate memory structure format details from the algorithms themselves, to provide some modularization or pluggability of memory formats.

3.3.2 Non-moving Mode

A simple memory manager (garbage collector) is provided which performs a mark-sweep-compact series of phases, with a tracked root-set and a card-marking mechanism to allow for low-level references into the heap to be handled safely and efficiently (objects marked will not be re-located automatically). This mode is as simple as it is for debugging and embedding simplicity.

3.4 Interaction Structures

3.4.1 External Resource Handles

3.4.2 FFI

4 C Translator and Dialect

4.1 Overview

A dialect is defined for translation of basic Slate programs which have a low-level semantics into safe, understandable, almost-idiomatic C programming language source code for quick portability across platforms.

4.2 Conventions

- Method definitions are not evaluated, just parsed and translated. Everything else is evaluated with a special treatment for type annotations.
- Prototypes are converted into C-structure types, using slots to specify structure elements and their types.
- Traits-installed slots are converted into globals, with the traits-name prepended to the name.
- Namespace-installed slots are converted into globals.
- Traits-installed immutable slots are converted into (symbolic) constants.
- Unary messages sent are translated into direct structure access wherever it is appropriate.
- Method names are "flattened" into C function names. Keyword selectors wind up with underscores substituted for colons, argument names transferred, and dispatches are handled by inserting the type's name at the appropriate position before an underscore. Binary selectors defined in the source are translated into English names with the same scheme about argument dispatches. For reference, calling `C SimpleGenerator cFunctionNameFor: selector on:` roles will answer the target name.
- Inlining occurs on a simple heuristic per method definition. The heuristic is that any method with a parse-node count of less than 10 and call count of less than 200 is immediately inlined. An annotation mechanism for the user to assist is also provided.
- Type annotations are limited. Basic C integer types may be attributed, or SmallInt direct values or pointers to structure type instances. These type annotations on expressions and explicit `cast` calls following type-annotations are changed into casts that comply with this scheme.

4.3 Restrictions

- Blocks may be used, but currently they may not be passed around as arguments to arbitrary methods.
- Inheritance is single-parent and static.
- Definitions are collected into modules to help determine consistency before generation.
- Dispatch is static. The dispatch must resolve to the appropriate C function name to result from the translation. If there is any ambiguity, an error should be raised during compilation for now.
- All definitions are statically type-checked, although simple automatic inference assists in minimizing the necessary impact of type annotations.

4.4 Special Methods

4.4.1 Arithmetic / Logical Operators

`+,-,*,/` generate C arithmetic operators. Self-assignments of any kind using appropriate math operators will generate operator-assignments in C.

`bitAnd:,bitOr:,bitNot` generate bitwise logical operators.

`min:,max:` generate conditional calls to return minimum/maximum of arguments.

4.4.2 Conditionals

`ifTrue:,ifFalse:,ifTrue:ifFalse:` generate appropriate if-then-else statements, or conditional expressions if the source expression is embedded in another.

`caseOf:,caseOf:otherwise:` generate an appropriate case-statement.

`isNil,isNotNil,ifNil:,ifNotNil:,ifNil:ifNotNil:` combine a null-test with conditionals.

4.4.3 Iteration

`to:by:do:,upTo:do:,downTo:do:,below:do:,above:do:` generate appropriate arithmetic-iteration for-loops.

`whileTrue:,whileFalse:,whileTrue,whileFalse,loop` generate appropriate while loops.

`break` sent to the current context will generate a loop iteration-breaking statement.

4.4.4 Memory Access

`address` generates a C address-of operator.

`load` generates a C dereferencing operator.

`store:` generates an assignment to the target of a pointer.

`cast` sent to a type-annotated expression to perform a C-style cast.

`at:,at:put:` generate a C array access/assignment.

`longAt:,longAt:put:` generate a word-oriented array access/assignment.

Table 1: Lexical Context Format

Word Offset	Interpretation	Type
0	Block	Method or Block
1	Frame	Frame Pointer (or Nil)
2	Variable 0	
2 + N - 1	Variable N - 1	

Table 2: Frame Format

Word Offset	Interpretation	Type
-N	Return Instruction Pointer	Fixed Integer
-N + 1	Variable 0	
-1	Variable N - 1	
0	Previous Frame	Fixed Pointer
1	Currently Executing Block	Method or Block
2	Lexical Context	Pointer (or Nil)

5 Bytecode Execution Engine

5.1 Context Management

5.1.1 Stack Format

The stack is an object array, which grows upward.

NOTE: Input variables precede local variables in stack frame.

Method format inherits fields from block format, prefixed by a pointer to the Selector (a pointer to a Symbol).

5.1.2 Establishment

1. Overwrite selector with return instruction pointer on stack.

Table 3: Block Format

Word	Interpretation	Type
0	Input Variables Count	Fixed Integer
1	Local Variables Count	Fixed Integer
2	Free Variables Count	Fixed Integer
3	Environment	(Pointer to) Namespace
4	Lexical Window	Array of Free Variable Arrays or Nil
5	Literal Array	Array of Literals or Nil
6	Selector Array	Array of Symbols or Nil
7	Code Array	ByteArray of Instructions
8	Syntax Tree	SyntaxNode or Nil

2. Allocate space for variables on stack.

3. Put current stack pointer in frame pointer.

4. Push previous frame pointer on stack.

5. Push block or method on stack.

6. If block needs free variables allocated,
   allocate lexical context and push on stack
   otherwise push Nil on stack.

5.1.3 Disestablishment

1. If lexical context is not Nil ,
   set frame pointer in lexical context to Nil.

2. Temporarily save top of stack as result value.

3. Set stack pointer to frame pointer.

4. Set frame pointer to previous frame pointer.

5. Pop return instruction pointer from stack into instruction pointer.

6. Push result value on stack.

5.2 Primitives

Opcodes have two different schemes for format. Normal opcodes are decoded by the bottom 4 bits of the byte, and use the top 4 bits to encode an immediate value argument. Extended opcodes have `0xF` in the low-order bits, and encode the operation in the high-order bits.

Immediate values are encoded as follows: if the value is less than 14, it is stored directly in the high-order 4 bits of the opcode byte. If not, the high-order bits are set to `0xF` (15) and each next byte is added to that value through the first non-filled byte (the first value not having its high bit set, i.e. `x bitAnd: 128/0x80`); the result is the immediate value.

5.3 Native Methods

Slate primitive bytecodes are deliberately restricted to control-flow: stack operations, jumps, and messaging primitives for sending and resending. All other core primitive methods are termed native methods, and only operate on heap objects. Core native methods are currently defined within the virtual machine, but it is intended that any part of the runtime be able to supply its own native methods.

Table 4: Normal Opcodes

Opcode	Name	Immediate Value	Interpretation
0	Invoke Message	Argument Count	
1	Load Variable	Variable Index	
2	Store Variable	Variable Index	
3	Load Free Variable	Lexical Offset	Next Byte: Free Variable Index
4	Store Free Variable	Lexical Offset	Next Byte: Free Variable Index
5	Load Literal	Literal Index	
6	Load Selector	Selector Index	
7	Pop	Pop Count	
8	Push Array	Array Size	
9	New Block	Literal Block Index	
A	Branch Keyed	Literal Table Index	
B	Message Invocation w/Optionals	Argument Count	
C	Non-Local Return	Lexical Offset	
D	Push Integer	Integer	
E	(unused)		
F	Extended...	Extended Opcode	

Table 5: Extended Opcodes

Opcode	Name	Interpretation
0	Jump	16-bit signed displacement follows
1	Branch If True	16-bit signed displacement follows
2	Branch If False	16-bit signed displacement follows
3	Push Environment	
4	Resend	
5	Push Nil	
6	Identity Equals	
7	Push True	
8	Push False	

6 Optimizing Compiler

6.1 Intermediate Representation

6.2 Machine Representation

6.3 Optimizer

6.4 Back-End