Matisse – A very brief and informal draft specification

Sven Karlsson

June 30, 2015

1 Items not yet described in this specification

- Code verification is not described.
- Exception processing is not described.
- The ISA is not complete.
- The DWARF extensions utilized by Matisse are not described.
- ELF header flags used by Matisse are not described.
- The garbage collector interface is not described.
- The runtime interface is not described.
- The binary formats of information tables are not described.
- The meta-data kinds are not described.
- Inter-executable communication is planned but not described.

2 Introduction

This document is a very brief specification of a new abstract machine, Matisse. The main goals of the abstract machine are to:

- provide a generic abstract machine which can handle a multitude of programming languages and programming paradigms.
- provide low enough primitives for system programming, yet and by default provide a secure high-level programming environment.

3 Preliminaries

The machine use ELF as executable binary format and DWARF [**?**] as debugging format and type information format. It is assumed that standard ELF mechanisms such as paging can be used.

The abstract machine leverages garbage collection.

4 Object model

DWARF type information is used to describe the storage layout of heap objects.

A machine implementation must store, at the very start of the heap object, a reference to type information in each heap object. The layout of this type information is machine specific.

The type information must be able to store a reference to an AnyType. The reference is set via a runtime sub-routine and can be read via an instruction. It is assumed to be used to implement constructs such as class methods.

The abstract machine delegates allocation of heap objects to the garbage collector but performs initialization. Heap objects can be write protected after initialization. In addition, heap object can be of the following two types: managed and un-managed. Un-managed heap objects are used for legacy imperative languages which require pessimistic garbage collection.

5 Execution file format

ELF is used as execution format. More precisely 32-bit ELF. All executables are position independent executables. The largest supported executable size is 2 gigabyte. Standard ELF mechanisms are supported including support for dynamic paging. Dynamic linking is supported, however it is not clear at this point what mechanism will be used for dynamic linking.

All executables are byte addressed.

As defined by the ELF standard, there is a single entry point in each ELF image. The entry point points to a non-returning sub-routine taking no parameters.

The least permissable access modes must be set on each ELF segment.

The executable segments consists of a set of subroutines which all must be verifiable. Each subroutine must have a statically determined maximal stack usage. Each subroutine must be fully contained in an ELF section with executable permissions. The ELF section index is used to identify the subroutine. Only one subroutine is allowed in a section and execution for each subroutine starts at the beginning of the respective section. Execution is not allowed to continue outside the section.

5.1 Symbol information

Executables have a language agnostic runtime at their disposal. Some runtime symbols are exposed to executables. To differenciate name spaces between the execution runtime and executables themselves, all language implementations must mangle their symbol names without exception. All symbol names are Modified UTF-8 strings with a single terminating byte with value 0.

Language implementations must prepend all their generated symbol names with the `'_'` character, single byte with decimal value 95. The same value is an illegal starting character for runtime symbols.

Symbol names are case sensitive. Language implementations must convert case insensitive names to lower case for interoperability.

5.2 Name mangling rules

TBA

6 Code verification

TBA

7 Execution model

Executables are defined by a set of ELF images. Executables may be multi-threaded. Each thread has a single stack consisting of stack elements. Each stack element has a declared type and an actual type. Stack sizes are not limited. Implementations may implement stacks as specially typed heap objects and dynamically allocate more stack space as needed.

Execution of an executable start with a single thread and with a stack with only a single activation frame. All stacks a private. Any shared stack, prescribed by the language implemented, must be implemented via heap objects.

Multiple executables may execute simultaneously. Executables may force the machine implementation to disallow multiple executables. This is indicated by a flag bit in the ELF header.

Task and transaction semantics is supported. A transaction is considered executed by the thread starting it. A spawned task is independent of the spawning thread and may be executed by the spawning thread or some other thread. However, tasks will not migrate between threads so the thread which started execution of a task will also complete the task.

Tasks do not share stacks with the spawning thread, spawning task, or any other task. Implementations may, however, execute tasks using the stack of the executing thread. The executed task is not allowed to access any stack space beyond its initial set of stack elements. The implementation must throw an exception if any such offending access is attempted.

Unsafe code, used to implement legacy languages with direct pointer manipulation capabilities such as C, is permitted in some situations. For unsafe code to be permitted all simultaneously executing executables and any dynamically loaded libraries they make use of must permit unsafe code. This is indicated by a flag bit in the ELF header.

Not permitted unsafe code cause an exception before any unsafe action is taken.

In addition, executables which permit unsafe code can via the runtime dynamically turn on or off exception generation for unsafe code. Hence, executables may dynamically disable safe code.

Memory operations take place in the order defined by the executable code and is, by default, only observable from the issuing thread context. However, the abstract machine may reorder memory operations or remove operations as long as program behavior is maintained. A rich set of memory fences is defined making it possible for the language implementation to define inter-thread memory consistency.

Finally, a complete set of atomic operations and synchronization primitives is provided.

8 Instruction set

The executable code is a stream of 8-bit unsigned bytes. However, in certain situation individual bytes are interpreted as a signed two-complement 8-bit quantity. Furthermore, 4 byte sequences are in some situations interpreted as big endian 32-bit unsigned words or big endian 32-bit signed two-complement words.

The executable code stream is broken down into instruction sequences consisting of a variable number of bytes. A instruction sequence start with a number of optional prefixes, followed by a single instruction followed by a number of optional suffixes.

Most instruction sequences may throw exceptions. Information about exception handlers is stored in the executables. Execution handler landing pads are static but a language implementation may implement its own dynamic exception scheme on top of the static scheme provided by the machine.

8.1 Information tables

The executable ELF format include a set of information tables. Each of these tables include a static set of entries with consecutive unsigned indexes ranging from 0 and upwards. Instructions refer to the tables via indexes encoded into the instructions. The tables are stored in Matisse specific ELF sections and the number of elements in the tables are inferred from the ELF section header.

The tables are:

constants Binary representations for constants used to initialize storage locations. The constant table is stored in the `.matisse.constants` section.

strings Constant, immutable, strings represented as Modified UTF-8 strings with a single terminating byte with value 0. The strings table is stored in the `.matisse.strings` section.

metadata Meta-data used to define types, sub-routines and other machine primitive which requires meta-data. The meta-data is stored in DWARF-4 format with matisse extensions. Meta-data kinds are:

typedefinition

typereference

subroutinedefinition

subroutinereference

fielddefinition

fieldreference

typealias

8.2 Machine types

The abstract machine has a number of built-in types which language implementations can use for mapping their types on. The types are broken down into: simple types, arrays, reference types, composite types and meta types:

Simple types

AnyType Dynamic type which at run-time can take on any possible type. Machine implementations are permitted to implement support large types for AnyType via boxed variables.

Boolean Single boolean value capable on taking either "true" or "false" values.

Int8 Signed two-complement 8-bit value.

UInt8 Unsigned 8-bit value.

Int16 Signed two-complement 16-bit value.

UInt16 Unsigned 16-bit value.

Int32 Signed two-complement 32-bit value.

UInt32 Unsigned 32-bit value.

Int64 Signed two-complement 64-bit value.

UInt64 Unsigned 64-bit value.

Ieee32 An floating point value in the standard IEEE 753-1985 32-bit representation.

Ieee64 An floating point value in the standard IEEE 754-1985 64-bit representation.

Address An unsigned integer value capable of representing an address. This type can only be used by unsafe code. Language implementations must not make assumptions about a certain number of bits being used for addresses. Conversions between integers and addresses will throw an exception if the source value can not be represented in the destination type.

Arrays

StaticArray <t><lowerLimit><higherLimit> A static contiguous array from integer index lowerLimit to higherLimit, inclusive. The number of encoding bits for the limits are encoded in the limit and must match one of the integer types. Each element in the array is of type <t>. Multi-dimensional arrays are permitted.

DynamicArray <t> A contiguous array with integer index. Each element is of type <t>and multi-dimensional arrays are allowed. The limits are not known statically at compile time. Instead the limits are set when the array is instanced and stored with the array.

Reference types

Reference <t> A reference to a heap object of type <t>. A Reference can only be converted to other Reference if the corresponding run-time types are consistent.

ArgumentReference A reference to an argument. It is used to implement variable argument calling conventions. An ArgumentReference cannot be converted into any other type. A number of runtime operations operate on ArgumentReferences allowing language implementations to safely iterate over arguments.

CodeReference <signature> A reference to the start of a subroutine which implements signature <signature>. A CodeReference cannot be converted to any other type. CodeReferences can only be created from a subroutinedefinition or a subroutinereference.

StackReference <t> A reference to the value of a stack element. StackReferences can never be boxed nor can types containing StackReferences be boxed. It is illegal to pass StackReferences as arguments to tasks.

Composite types

Composite <t+> Composite type with members each having their own type. Any type is allowed as a type of a member but not an Composite type. Each member has a name, versions, type and a set of properties. Subroutines, when members, are defined with a number of of subroutinedefinition or subroutinereference and a set of properties. Fields, when members, are defined with a number of fielddefinition or fieldreference and a set of properties.

Composite type may refer to other Composite types by having type aliases as members

Composite types themselves have names, versions and can have properties.

Types for describing types or metadata

Type An opaque machine implementation specific type describing a type. The purpose is to allow language implementations to operate on types, for example to dynamically create stack elements with types not known at compile time.

Metadata An opaque machine implementation specific type describing metadata. The purpose is to allow language implementations to operate on metadata, for example to facilitate reflection.

All type names are case sensitive. An compliant machine implementation may decide to use more bits than necessary to implement each type. Language implementations must not assume that types are being implemented with a precise number of bits.

8.3 Utility data structures and types

Many instructions make use of the `PCRel32` data type which is a relative address only existing in instruction encodings themselves. Instructions can never operate on PCRel32 quantities themselves. PCRel32 is a big-ending 32-bit two-complement value. It points to a location in a loaded readonly ELF segment by

taking the absolute address of the first byte in the PCRel32 value and adding the PCRel32 value.

In addition, several data structures are laid out in loaded and readonly ELF segments.

8.4 Instructions listing

The following is an authorative list of prefixes, suffixes and instructions with decimal byte values. Many prefixes and suffixes are composable. Some prefixes also composable with suffixes. All mnemonics are case sensitive.

8.5 Prefixes

`volatile`**, 255, composable, composable with suffixes** The succeeding memory operation must be carried out and must be carried out in program order.

`unsafe`**, 254, composable but must directly proceed instruction** Switches, for the duration of one instruction, the instruction set to the unsafe instruction set, see section 8.8.

`noOverflow`**, 253, composable** Enables overflow exceptions for the duration of one instruction.

`large`**, 252, composable** Succeeding instruction argument constants are big endian 32-bit words for the duration of one instruction.

`suffixExist`**, 251, composable** Succeeding instruction is trailed by a number of suffixes.

8.6 Suffixes

8.7 Instructions

NOTE: Indirect versions require a metadata type in the top of stack which will describe a member, type or subroutine.

`pushElement <UInt8 displacement>`**, 0** Instruction byte is followed by a single byte constant interpreted as an unsigned 8-bit integer. The integer is a displacement upwards into the stack with the top of stack being displacement 0. The stack element pointed to by the displacement in pushed onto the stack preserving both declared and actual types of the pushed element. An exception will be thrown if the displacement points to the activation element, outside the stack or beyond the last argument to the sub-routine. The large prefix will extend the displacement argument into a big-endian UInt32 and so will take up 4 bytes.

`pop <UInt8 elements>`**, 1** Instruction byte is followed by a single byte constant interpreted as an unsigned 8-bit integer. The integer holds the number of stack elements to remove from the stack. A value of 0 is illegal and will cause an exception. An exception will be thrown if the activation element is removed. The large prefix will extend the elements argument into a big-endian UInt32 and so will take up 4 bytes.

`storeElement <UInt8 displacement>`, **2** Instruction byte is followed by a single byte interpreted as an unsigned 8-bit integer. The integer is a displacement upwards into the stack with the top of stack being displacement 0. The top of stack popped from the stack and then stored into the stack element pointed to by the displacement. The actual type of the stored stack elements must be consistent with the declared type of the store location. An exception will be thrown if the displacement is 0, if types are not consistent, if it points to the activation element, outside the stack or beyond the last argument to the sub-routine. The large prefix will extend the displacement argument into a big-endian UInt32 and so will take up 4 bytes.

`return`, **3** Removes all stack elements up until the activation element then returns to the sub-routine pointed to by the activation element and removes the activation element.

`pushLiteral <UInt8 typeRepresentation, Int8 value>`, **4** Push a literal to the stack. The value of the literal is defined by the value operand while the type of the created stack element is defined by the typeRepresentation operand which is an index into the meta data table and must correspond do a typedefinition or typereference. Literals not representable with an 8 bit or 32 bit value can be pushed on the stack with pushConstant.

`pushConstant <UInt8 typeRepresentation, UInt8 constantRepresentation>`, **5** Push a constnat to the stack. The value of the constant is defined by the constantRepresentation operand which is an index into the constant table. The type of the created stack element is defined by the typeRepresentation operand which is an index into the meta data table and must correspond do a typedefinition or typereference.

`pushMeta <UInt8 index>`, **6**

`pushField <UInt8 index>`, **7**

`popField <UInt8 index>`, **8**

`pushFieldIndirect`,

`popFieldIndirect`,

`pushFieldHeapObject <UInt8 index>`,

`popFieldHeapObject <UInt8 index>`,

`pushFieldIndirectHeapObject`,

`popFieldIndirectHeapObject`,

`pushFieldStackObject <UInt8 index>`,

`popFieldStackObject <UInt8 index>`,

`pushFieldIndirectStackObject`,

`popFieldIndirectStackObject`,

`pushInitialized <UInt8 typeRepresentation, UInt8 argumentCount>`,

`pushInitializedIndirect <UInt8 argumentCount>`,

`pushUnInitialized <UInt8 typeRepresentation>`,

`pushUnInitializedIndirect`,

`new <UInt8 typeRepresentation, UInt8 argumentCount>`,

`newIndirect <UInt8 argumentCount>`,

`throw`,

`box`, Pops the top of stack and convert the stack element into a heap object whose type is the actual type of the stack element. A reference to the heap object, typed as Reference <declared type of the stack element>, is pushed onto the stack. An exception will be thrown if the top of stack cannot be converted into a heap object, such as the activation element The declared and actual type of the pushed reference is identical.

`unbox`, The top of stack must be a Reference. Pops the top of stack and convert the heap object pointed to by the Reference into a stack element. The stack element is then pushed on the stack. The declared type of the created stack element is the type pointed to by the Reference. The actual type is the type of the heap object converted. After the operation, the heap object may live on and so the stack element is a copy of the heap object at the time `unbox` was executed.

`referenceStackElement <UInt8 displacement>`, Instruction byte is followed by a single byte constant interpreted as an unsigned 8-bit integer. The integer is a displacement upwards into the stack with the top of stack being displacement 0. A StackReference pointing to the stack element pointed to by the displacement is pushed onto the stack. The declared type of the pushed StackReference is StackReference <declared type of the referenced stack element>and the actual type is StackReference <actual type of the referenced stack element>. An exception will be thrown if the displacement points to the activation element, outside the stack or beyond the last argument to the sub-routine. The large prefix will extend the displacement argument into a big-endian UInt32 and so will take up 4 bytes.

`call <PCRel32 offset>`,

`invoke <UInt8 subroutineRepresentation, UInt8 argumentCount>`,

`invokeIndirect <UInt8 argumentCount>`,

`invokeHeapObject <UInt8 subroutineRepresentation, UInt8 argumentCount>`,

`invokeIndirectHeapObject <UInt8 argumentCount>`,

`invokeStackObject <UInt8 subroutineRepresentation, UInt8 argumentCount>`,

invokeIndirectStackObject <UInt8 argumentCount>,

convert <UInt8 typeRepresentation>,

convertIndirect,

add,

addSaturated,

subtract,

subtractSaturated,

negate,

negateSaturate,

and,

or,

xor,

not,

8.8 Unsafe instructions

load <UInt8 typeRepresentation> store <UInt8 typeRepresentation>

8.9 Instructions for transactions

8.10 Tasking instructions

8.11 Instruction for atomicity

8.12 Memory fences

References