GFR-1

The Gear Virtual Machine

Version 0.1

Markéta Lisová

May 16, 2015

Contents

Preface

The *Gear Virtual Machine* (the Gear VM, or GVM) is a specification for a set of programs and program components. Those programs should enable the Gear programming language (and possibly many more) to execute on virtually any *host platform* that the Gear VM is implemented on.

Status of This GFR

This GFR is active and mandatory for every proper Gear implementation that includes and targets a Gear VM. The status of this GFR is not likely to change. This GFR does not prevent existence of Gear implementations that target a different runtime environment, perhaps using a subset or superset of the Gear language. Gear bytecode is however intended to be compatible with a Gear VM primarily, without regard of any other pre-existing or post-existing runtime environment.

Chapter 1

Abstraction vs. the Platform

Like Java VM, a Gear VM should shield users of its programming languages from specifics of each platform as much as possible. E.g. numbers – each platform has specific representations of numbers, some of which may fit into the languages, some may need to be represented in a more virtual form, like the `Decimal` in Gear, which represents virtually any non-complex number digit-by-digit.

However, users may need to access the platform natively, in some limited program scope. For that purpose, a Gear VM should provide a native interface, which would inherently be specific to the platform and to the technology that a Gear VM is built with, but most likely it could be just a set of bindings written in the C language.

A Gear VM has to also provide ways for inter-program communication. A special channel may be available for Gear VM-to-Gear VM communications, and platform's standard channels should also be available, e.g. sockets, shared memory, or messaging services. The range of available non-Gear VM-to-Gear VM channels is inherently limited by the host platform and implementation status of a Gear VM.

1.1 Bytecode Portability

A single bytecode format is required for any Gear VM. A file containing such a bytecode must be executable by any Gear VM, regardless of the host platform. This single bytecode format is derived from the needs of the Gear programming language, and is tagged with a *dialect* – basically a short identifier of the language that is supposed to serve as its backend. Therefore, the single bytecode format is portable to any host platform that has the backing language installed.

It is however possible for the language to define a Gear VM extension, that would be able to transform any bytecode format it needs into a Gear VM-compatible one. Such extension should be portable between each host platform.

If a program that targets a Gear VM is to contain native code alongside with Gear VM bytecode, then such program is limited to the host platforms that the native code is compatible with. This native code could be distributed in these major ways:

- A dynamic library to be loaded at runtime as an extension for a Gear VM. A Gear VM then provides a runtime environment for the library.
- A separate executable to be executed as a separate process from a Gear VM, possibly from within a shell. Such an executable is executed in the context of the host platform, not a Gear VM.

1.2 Host Platforms

The initially supported host platforms should be these:

- Mac OS X (initially 10.9+)
- Windows (initially 8.1+)
- GNU/Linux and BSD (initially only recent versions)

More platforms are indeed welcome to become supported, but these are the main ones.

Chapter 2

The Abstraction

With abstraction, we here have the concepts of human-to-computer and language-to-computer communication in mind. Users will use a programming language of choice to communicate with a host platform, in a way that is friendly to them and independent of the host platform, except where needed.

2.1 The Languages

Initially, the Gear programming language is the primary supported language and may also serve as a intermediary between any further supported languages, using its own data types and instruction sequences.

A language definition for a Gear VM is basically a set of callbacks to be invoked for various reasons, e.g. method lookup, PSI or AST manipulations and so on. A language should also contain a compiler, which will provide a way to transform a source file set written using it into a form that is understandable by a Gear VM.

A Gear VM will provide primarily constructs to support the Gear programming language, but it may also be extended in the future to add more constructs for other languages, if ever needed.

A Gear VM should support programs written in several languages by keeping track of origins of values – thus each language may use e.g. different object layouts, and yet still make its values available to objects coming from other languages via Gear VM's shared protocols. Such tracking should be implemented using a small tracking label, possibly a number mapped to loaded language definitions.

For the supported language, as Gear is a hybrid language (it could be seen as both an object-oriented and a non-purely functional), it would be nice if a ML-based or a Lisp-based functional language was supported as well. It might be even possible to create a

trans-compiler from other pre-existing languages into Gear VM bytecode, e.g. Java, C# or even C. However, such pre-existing languages would need their backing language definition for a Gear VM to use.

2.2 The Bytecodes

A Gear VM should natively support only a single bytecode format – the Gear VM bytecode. Other bytecode formats may be supported by means of Gear VM extensions, working as a compiler from a custom bytecode into a Gear VM bytecode, primarily to support sort of a transpilation, e.g. from Java's `.class` format directly into the Gear VM bytecode format, if such extension is ever built.

The Gear VM bytecode format has to be complex and robust enough to accommodate most languages' needs.

Chapter 3

The Gear VM Bytecode

The Gear VM bytecode is a binary file containing several parts:

- Metadata – the dialect (i.e. the backing language), bytecode version etc.
- Constants – e.g. number literals, strings, symbols, type paths.
- Instruction sequence graph – graph structures of instruction sequences, one per function. Such a graph is represented by a serialized sequence of instruction nodes along with links to other nodes as specified by each node. The original source file may be a single instruction sequence graph, as it is with Gear.

This chapter describes the Gear VM Bytecode file format, `.gearb`. Each `.gearb` file contains definition of a main instruction sequence graph along with additional graphs, which is unlike Java's **`.class`** file format, which only contains definition of a single class.

A `.gearb` file consists of a stream of 8-bit bytes. All 16-bit, 32-bit, 64-bit and 128-bit quantities are constructed by reading in two, four, eight, and sixteen consecutive 8-bit bytes, respectively. Multibyte data items are always stored in big-endian order, where the high bytes come first.

The types `u1`, `u2`, `u4`, `u8` and `u16` represent an unsigned one-, two-, four-, eight-, or sixteen-byte quantity, respectively.

In order to save disk space, numbers can be also expressed with multi-byte constructs. The types `m`n, where $1 \leq n \leq 20$, then represent 8-bit byte sequences in big-endian order, where only the last 7 bits are used to represent the corresponding number and the very first bit indicates whether the byte is the last byte in a sequence (the bit is `0`), or if it does not end a sequence (the bit is `1`). The types `m1`, `m2`, `m3` etc. then respectively represent unsigned numbers of 7-bit, 14-bit, 21-bit. The pseudo-type `mx` then represents any of these types.

The `.gearb` file format is presented here using pseudo-structures, written in a C-like notation. The contents of these structures are referred to as *items*. An item may also be a *table*, which consists of zero or more variable-sized items. Due to that fact, a table index may not be used to directly reference a byte offset in the file.

3.1 The `File` Structure

A `.gearb` file consists of a single `File` structure:

```
File {
  u4 magic;
  u4 major_version;
  u4 minor_version;
  dialect_info dialect;
  mx constant_pool_count;
  cp_item constant_pool[constant_pool_count];
  mx graphs_count;
  graph graphs[graphs_count];
}
```

The items in the `File` structure are as follows:

`magic`
: The `magic` item supplies the magic number identifying the `.gearb` file format; for the Gear VM bytecode, it has the value of `0xC0124189`.

`major_version, minor_version`
: The `major_version` along with the following `minor_version` items determine the version of the bytecode file format. A particular Gear VM may only support file formats in a certain range.

`dialect`
: The `dialect` item specifies the identifier and version of the backing language of the `File` structure.

`constant_pool_count`
: The value of the `constant_pool_count` item is equal to the number of entries in the `constant_pool` table.

`constant_pool`
: The `constant_pool` item is a table of structures representing various string constants, symbols, numbers and other constants that are referred to within the `File` structure and its substructures. Each entry in it is tagged with its format. The `constant_pool` is indexed from `1` to `constant_pool_count`.

graphs_count
: The value of the graphs_count item is equal to the number of entries in the graphs table.

graphs
: The graphs item is a table of structures representing instruction sequence graphs. The graphs table is indexed from 1 to graphs_count.

3.2 The constant_pool Table

Instructions refer to items in the constant_pool table. All entries have the following general format:

```
cp_item {
  u1 tag;
  u1 data[];
}
```

Each item in the constant_pool table must begin with a 1-byte tag indicating the kind of cp_item entry. The contents of the data array vary with the value of tag. The valid tags and their values are listed in ??. The format of the additional information in data varies with the tag value.

Table 3.1: Constant pool item tags

Constant tag	Value
`Const_UTF8`	1
`Const_Type_Path_Mutable`	2
`Const_Type_Path_Immutable`	3
`Const_Int8`	4
`Const_Int16`	5
`Const_Int32`	6
`Const_Int64`	7
`Const_Int128`	8
`Const_UInt8`	9
`Const_UInt16`	10
`Const_UInt32`	11
`Const_UInt64`	12
`Const_UInt128`	13
`Const_Float32`	14
`Const_Float64`	15
`Const_Float128`	16
`Const_Decimal`	17
`Const_Complex`	18
`Const_Symbol`	19
`Const_String`	20

3.2.1 The `Const_UTF8_Item` Structure

A Gear VM must use the unmodified UTF-8 format. It is an universal item for text representation, used in both strings and symbols.

The `Const_UTF8_Item` structure is:

```
Const_UTF8_Item {
  u1 tag;
  mx length;
  u1 bytes[length];
}
```

The items of the `Const_UTF8_Item` are the following:

`tag` The `tag` item of the `Const_UTF8_Item` structure has the value `Const_UTF8` (1).

`length`
: The value of the `length` item gives the number of bytes in the `bytes` array (not the length of the resulting string or symbol, due to the nature of multi-byte UTF-8).

It is not null-terminated, as that would be redundant. If the array ends with a null byte, it is a real part of the data.

`bytes`
: The `bytes` array contains the bytes of the string. The bytes may have any value.

3.2.2 The `Const_Type_Path_*_Item` Structures

A type path is a simple list structure that consists of:

- A variable symbol.
- A selection symbol.
- An application type path argument.
- A type application type path argument.
- An application comma.

The `Const_Type_Path_Mutable_Item` (and `Const_Type_Path_Immutable_Item`[1]) structures are:

```
Const_Type_Path_Mutable_Item {
  u1 tag;
  mx length;
  mx element_count;
  u1 flags;
  u1 element_types[element_count];
  u1 elements[length];
}
```

The items of both the `Const_Type_Path_Mutable_Item` and `Const_Type_Path_Immutable_Item` are the following:

`tag` The `tag` item of the structure has the value `Const_Type_Path_Mutable` (2) or `Const_Type_Path_Immutable` (3).

`length`
: The value of the `length` item gives the number of bytes in the `elements` array (not the length of the resulting type path).

`element_count`
: The value of the `element_count` item gives the number of bytes in the `element_types` array.

[1]The structures are the same when it comes to their format.

`element_types`

The `element_types` array contains bytes that mark the type of each element in the `elements` array. The valid values of these bytes are listed in ??. This array is necessary, because there is no possible sub-language that would express a type path with a special syntax within a single byte array, because a Gear VM supports identifiers made by any UTF-8 strings.

`elements`

The `elements` array contains bytes that are in fact multi-byte numbers (the `mx` type). Indices from `element_types` then refer to the assembled `mx` types, not each byte. Values of these numbers are then references to `Const_Symbol_Items` (variables, selections) or another `Const_Type_Path_Items` (applications, type applications, expected type). It is an error if a type path contains more that one expected type element, and it is okay if it does not contain any expected type (the expected type is then undefined). The index of the expected type element within the `elements` array is unspecified. There are no bytes for elements that are a comma, as that is expressed by the type already. A zero value of an element signifies an unspecified type element.

Table 3.2: Type path flags

Flag	Value	Note
`Type_Path_Relative`	0	path is relative
`Type_Path_Rooted`	1	path is absolute
`Type_Path_Private`	2	path is not referring to user-accessible type
`Type_Path_Null_Override`	4	path is overriding nullability
`Type_Path_Nullable`	8	path is forced nullable; 0 on this bit is forced non-nullable

Table 3.3: Type path element types

Element type	Value	Note
Type_Path_Variable	1	expects index of a symbol
Type_Path_Selection	2	expects index of a symbol or type path
Type_Path_Application	3	expects index of a type path
Type_Path_Type_Application	4	expects index of a type path
Type_Path_Expected_Type	5	expects index of a type path
Type_Path_Left_Parens	6	not present in elements
Type_Path_Right_Parens	7	not present in elements
Type_Path_Application_Name	8	expects index of a symbol
Type_Path_Projection	9	not present in elements
Type_Path_Rooted	10	not present in elements
Type_Path_Splat_Flag	64	not present in elements, combined with other values
Type_Path_Implicit_Flag	128	not present in elements, combined with other values

3.2.2.1 Type Paths & Scopes

A type path—relatively to its used position and scope—determines a type, either by pointing it to a node of PSI (§??), or, with a variable symbol, to an actual type of a value in a variable. The list of possibilities of type paths may get extended with support for more backing languages.

A type path is built as a sequence of references to symbols in the symbols list and a sequence of designators for each of its node (determining whether a node is a variable symbol, a selection symbol etc.).

A type path can also refer to a type that is made available with a type scope, which includes mechanism that can mark a scope being a parent scope of another scope (this is important for reuse of type scopes – and type scopes can possibly be reused across different bytecode files as well). When talking Gear, this can be done by using **use** clauses to import names into a particular scope – such types are then available with type paths that start with a selection symbol.

A type scope can also have a wildcard import, by using an extension of a backing language. When talking Gear, a scope can be tagged with a list of types from which it is built up, along with their versions at the time that import was performed. Thus, Gear is able to detect a dynamic change in a scope and next time that scope is accessed, the language is given an opportunity to refresh the scope and check for duplicates (which would be an error).

A language has to define the order of search for a type, since a type scope is not the only place where it may look for. The way it defines that is irrelevant to Gear VM, and it does so by implementing the language bindings (§??).

- The type of the function itself, if the language supports higher-order function types and their extension.
- The type of receiver, if any.
- Super-type(s) of receiver, if any.
- The given type scope assembled while evaluating an instruction sequence graph.
- Another related types to the type of receiver (if any) or the type of the function, such as a namespace or a module.
- Use a dynamic callback.

To resolve a type that is present in the PSI (note that PSI contains both named and anonymous types), a language binding has to be invoked with the assembled type scope and the corresponding type path with variable types resolved automatically by a Gear VM.

The backing language then with its implementation defines the order of search (which is arbitrary to each backing language) and either returns a type from PSI, or signals that it could not find a type, which is an error state that is further resolved. Such resolution may be implemented directly in the instruction that failed as a fallback instruction, or if that fails too or is not set up, another language binding (e.g. in Ruby, that could lead to an invocation of `const_missing`).

A very interesting case is when a backing language implements implicit conversions. E.g., say that a type path contains a function application and for some (or maybe even all) of the argument types, the corresponding member is not applicable to the given argument types. The language can make use of an implicit scope – e.g. search the type scope for a type that can convert a particular argument type to a type that the corresponding member is applicable to, or search some another type for such a conversion.

To allow the aforementioned behavior, all types referenced from a type scope are tagged with an "implicit flag", which (when set) makes the type eligible for implicit search & querying.

A Gear VM helps the backing language by providing an API to look for an implicit conversion on demand (therefore, a particular backing language does not need to make sense of the implicit flags at all). A Gear VM also tells the language where does that conversion come from, so that the language may choose whether to use it or not. A Gear VM can also be instructed to only look into the type scope, or some arbitrary other type, to collect all implicit values filtered by the backing language's query.

With an implicit value for conversion found, the backing language may instruct a Gear VM to apply the conversions before applying the function that requires the conversions.

An implicit conversion based on an implicit value is not the only option of converting an argument type though – a backing language has also the following options of converting an argument type (or even the selection itself):

- An implicit value conversion on argument.
- An implicit value conversion on selection. [2]
- A conversion specified by the language itself (e.g. numeric widening).
- Application conversion. [3]
- Eta-expansion. [4]
- Type application. [5]

A language may indeed choose multiple such conversions (based on its rules) for each component of a type path, and even multiple conversions for one component (e.g. an implicit value conversion followed by an application conversion). It is therefore possible to even replace the whole type path with conversions and cache the resulting conversion (the key in a cache is the type path in the given used position).

When talking Gear, the language may also implement implicit parameters, along with implicit conversions. Such a feature may be rendered by using an application conversion on a modified type path, adding a flag that such an extra application is implicit (a Gear VM has to know of such possibility), and then what Gear VM does is: if the original function application is directly followed by another application (without selecting another member on the previous result), it ignores that application conversion added to the type path, and if not, then another function application happens, based on the newly added "virtual" application conversion (and yet again, the language is tasked with resolving each argument type, which is now tagged as implicit, so that it knows that it should only look for implicit values).

3.2.3 The `Const_(U)Int`n`_Item` Structures

A Gear VM provides for its bytecode a rather wide range of integral types for constants, including both signed and unsigned versions. Numbers that fall out of the scope of these types or numbers that have to be as precise as possible have the `Const_Decimal_Item` structure available (including real numbers).

[2] Used in Gear for view application.

[3] Used in Gear for method conversion – evaluation and empty application.

[4] Basically a conversion from a complex type path into a simple path that ends with an application symbol, created by evaluating the preceding nodes up to the last function reference. If the type path is an application, then it gets converted to a selection instead, so that incomplete applications to methods with multiple parameter lists can be converted into partial applications – and therefore the backing language does not need to care about that, as long as it registers its multi-param-list methods correctly along with ad-hoc lambda support (true for every backing language that has registered an appropriate function type).

[5] Used in Gear for value conversion – type instantiation, and also type inference that does not need to be deferred on selections, as the whole selection is made available with the type path, and Gear chooses to treat empty applications as selections.

The `Const_Int`n`_Item` structures are:

```
Const_Int8_Item {
  u1 tag;
  u1 value;
}
Const_Int16_Item {
  u1 tag;
  u2 value;
}
Const_Int32_Item {
  u1 tag;
  u4 value;
}
Const_Int64_Item {
  u1 tag;
  u8 value;
}
Const_Int128_Item {
  u1 tag;
  u16 value;
}
```

The `Const_UInt`n`_Item` structures are:

```
Const_UInt8_Item {
  u1 tag;
  u1 value;
}
Const_UInt16_Item {
  u1 tag;
  u2 value;
}
Const_UInt32_Item {
  u1 tag;
  u4 value;
}
Const_UInt64_Item {
  u1 tag;
  u8 value;
}
Const_UInt128_Item {
  u1 tag;
  u16 value;
}
```

The presence of a sign bit in each structure is determined by its tag: `Int`n items are signed, `UInt`n items are not signed. Unsigned versions of these structures have the same layout.

3.2.4 The `Const_Float`n`_Item` Structure

A Gear VM provides for its bytecode three structures for representing floating-point numbers in binary form. Each value is represented using the IEEE 754 n-bit precision format.

The `Const_Float`n`_Item` structures are:

```
Const_Float32_Item {
  u1 tag;
  u4 value;
}
Const_Float64_Item {
  u1 tag;
  u8 value;
}
Const_Float128_Item {
  u1 tag;
  u16 value;
}
```

3.2.5 The `Const_Decimal_Item` Structure

A Gear VM provides for its bytecode a structure for representing arbitrary-precision and arbitrary-size numbers, including integral and real numbers.

The `Const_Decimal_Item` structure is:

```
Const_Decimal_Item {
  u1 tag;
  mx length;
  mx scale;
  u1 numbers[length];
}
```

3.2.6 The `Const_Symbol_Item` Structure

A Gear VM uses this structure to represent names in programs.

The `Const_Symbol_Item` structure is:

```
Const_Symbol_Item {
  u1 tag;
  mx name_index;
}
```

3.2.7 The `Const_String_Item` Structure

A Gear VM uses this structure to represent text literals in programs.

The `Const_String_Item` structure is:

```
Const_String_Item {
  u1 tag;
  mx string_index;
}
```

3.2.8 The `Const_Complex_Item` Structure

A Gear VM uses this structure to represent complex numbers in programs.

The `Const_Complex_Item` structure is:

```
Const_Complex_Item {
  u1 tag;
  mx real_index;
  mx imaginary_index;
}
```

3.3 The `graphs` Table

The `graphs` table contains `graphs_count` items. Each item represents an instruction sequence graph and has the following general format:

```
graph {
  u1 opcode;
  u1 data[];
}
```

Each item in the `graphs` table must begin with a 1-byte opcode indicating the kind of `graph` entry. The contents of the `data` array vary with the value of `opcode`. The valid opcodes and their values are listed in ??. The format of the additional information in `data` varies with the `opcode` value.

Table 3.4: Graph opcodes

Opcode	Value
Op_Noop	1
Op_Sequence	2
Op_Store	3
Op_Store_Multi	4
Op_Branch	5
Op_Branch_Neg	6
Op_Parallel	7
Op_Jump	8
Op_Label	9
Op_Val_Dcl	10
Op_Param_Dcl	11
Op_Result_Dcl	12
Op_Unapply	13
Op_Apply	14
Op_Test_Type	15
Op_Use_Type	16
Op_Const	17
Op_Val	18
Op_Return	19
Op_Throw	20
Op_Catch	21
Op_Annotated	22
Op_Type_Cache	23
Op_Metadata	24
Op_Native	25
Op_Loop	26
Op_Memoize	27
Op_Self	28
Op_Super	29
Op_Nested_Self	30
Op_Nested_Super	31
Op_Select	32
Op_Type_Dcl	33
Op_Type_Def	34
Op_Method_Dcl	35
Op_Method_Def	36
Op_Type_Param_Dcl	37

Chapter 4

The Components of a Virtual Machine

4.1 Language Interface & Bindings

4.1.1 Compiler Interface

4.1.2 Runtime Interface

4.2 Command Line Interface

A command line interface to a Gear VM should enable two main important functions: to start a Gear VM with a specified program, and to cover compilation of source of programs and libraries.

4.3 Threads

A Gear VM runtime is all about *threads*. A thread models a computation stream, therefore computations may be parallel. Moreover, each thread may compose of multiple *fibers*, which are modelled using separate call stacks, some of which do not need to be complete copies. This also enables continuations, both unlimited and delimited.

A Gear VM should prefer native implementation of threads wherever possible, providing an abstraction to the languages, which in turn may provide abstraction to its users. Fibers, on the other hand, are not really easy to implement using native resources.

Fibers may not be passed between threads, but may be copied and then the copy may be transferred to a different thread, but only before it is resumed. This may happen on a language level as well as user level, or even both (as in Gear, where the language

enables passing of fibers between threads, but users have to make up the logic in which the receiving thread will pick up the fiber object).

4.4 Stacks

A *stack* is the low-level structure of a Gear VM. It contains a stack of stackable frames, which are structures containing reference to an instruction graph, maintaining an instruction pointer (program counter), containing references to argument values, the result value and thrown values, and optionally some more metadata, e.g. line numbers and corresponding files.

A stack may or may not represent a whole fiber execution, but there is always at least one stack per thread that does. The other, incomplete ones, are delimited and used with constructs such as other fibers and continuations.

A stackable frame internally increments reference count to the instruction sequence graph that it references, and decrements it again upon being popped off its stack. This way an instruction sequence graph is persisted during evaluation, while the original function may be already deleted or replaced after a hot swap of code.

A stackable frame may be a control frame, containing the aforementioned elements, or a native frame, filling in for a native method invocation via `FFI` (Foreign Function Invocation). A native frame does not need to contain any program counters due to the nature of its execution. The native function is given access to its stackable frame, so it can e.g. return or throw a value.

A Gear VM should use the so-called “saguaro stack”, or “parent pointer tree”, so that fibers are possible/easier to implement.

4.5 Address Spaces & Values

A Gear VM has to give its languages memory for allocating new objects. In the context of a Gear VM, an object and a value are referring to the same concept – it represents a value that has some address, properties and size.

Languages are given APIs to handle the memory that is managed by a Gear VM. Those APIs are implementation-defined, however. Some values may be represented as “immediates”[1], and a Gear VM shall handle transitions between immediates and referenced values transparently. To prevent undefined behaviours, both Gear VM and the languages are required to not use non-pure NaN values, since that could break immediates.

[1] An immediate may be e.g. embedding integral numbers, floating point numbers and/or other special values like booleans into a pointer, or some other combination, like NaN-coding.

Every object has its address, locating it in the memory. Every object that is allocated on "the heap" has to be either reference counted, or garbage collected. Automatic reference counting is preferred.

For reference counted memory management, every object is associated with a reference counter. It is implementation-defined how this association works. The only requirement of a Gear VM is to provide strong and weak references. Soft, phantom or other references are optional, and are not implemented in Standard Gear implementations. A Gear VM has to be able to handle at least 10000 strong references and equal number of weak references.

Objects are referred to via variable values, which are generally placeholders for object addresses. Addresses may be cached per-thread, unless the variable is specified to be volatile. Volatile variables must be only instance variables, and can opt-in for acquire/release semantics or sequential consistency.

An object can have any number of properties, and also a set of flags that are common throughout every language (e.g. the "frozen" state for immutable objects). This specification does not require any specific memory layout of object representation. An object may have at least two kinds of instance variables – those known at compile time, and those that are added in runtime dynamically. A Gear VM has to be able to handle at least 255 instance variables in memory layout of objects of any kind.

4.6 Program Structure Information

A program structure information (`PSI`) is a forest of graph structures that hold both compile-time and runtime information about types in a program. One of the most important data stored in it are method references.

Languages may opt in to implement class objects by having their objects point to various subgraphs of a PSI.

4.7 Inter-VM communication

A Gear VM has to implement a mechanism for inter-VM communication between programs. Furthermore, such VMs do not have to run on the same computer. The toolchain provided for this communication does not need to be embedded in a Gear VM and can be shipped separately.

Also a separate GFR is to be made for this component of a Gear VM.

4.8 Native Components

A Gear VM has to also provide guided access to some components that are underneath mapped to native host platform components, e.g. everything input/output or thread synchronization primitives. This is so that a language can be useful (input/output) and also parallelized (synchronization).