The Glasgow Virtual Machine Toolkit Manual

Mark Shannon

Abstract

The GVMT is a toolkit for building virtual machines. The GVMT does not dictate the overall design of the virtual machine, but it does eliminate a lot of implementation details.

The developer needs to develop an interpreter and supoorting code, as normal. The toolkit handles garbage collection transparently and can automatically create a just-in-time compiler from the interpreter source code.

This manual covers the components of the Glasgow Virtual Machine Toolkit, hereafter called the GVMT, and their use.

Convertion of source-code to bytecodes is not handled by the GVMT, that is up to the developer.

Contents

Chapter 1

Introduction

The Glasgow Virtual Machine Toolkit, GVMT, is a toolkit for constructing high-performance virtual machines.

The GVMT can be divided into three parts:

- Tools for the creation of interpreters, compilers and other virtual machine components.
- Library code. Primarily for Garbage Collection.
- An interface between the user-defined parts of the virtual machine and the toolkit library.

All the components operate on a common abstract machine model. It is not necessary to understand this model to use the GVMT, but a knowledge of the abstract machine is useful to get the best out of the GVMT. The abstract machine, which is described in detail in Chapter 3, is a stack-based machine supporting concurrency and garbage-collection. It has an extensible instruction set. The instruction set is abstract in that it cannot be directed executed, but is used to defined the semantics of bytecodes and other code.

The tools provided by the GVMT can be split into front-end tools which compile source code to the abstract machine model, and back-end tools which convert from the abstract machine code to real machine code.

1.1 Example virtual machine

In order to help understand the GVMT, a simple Scheme implementation is included as an example, it can be found in the `/example` directory. This example illustrates all the important aspects of the GVMT, while not being overly complex.

All scheme VMs are implemented by what is known as read-eval-print loop. Its reads the input, evaluates the input and then prints the result, repeating until the interpreter exits. The 'read' part consists of reading text from a file or terminal and parsing that text to form a syntax tree.

The file `/example/parser.c` contains the parser, it is a standard C(89) source file which can be compiled with the GVMT C compiler, GVMTC(see Section 2.1.3)

In some scheme implementations the 'eval' part is implemented by directly evaluating the syntax tree. This is not the case for the GVMT version, as all GVMT-based VM use bytecode interpreters. The GVMT scheme implementation first compiles the syntax tree to bytecode, then evaluates the resulting bytecode. The syntax-tree to bytecode compiler code can be found in `/example/compiler.c`.

The bytecode interpreter is defined in `/example/interpreter.vmc`

Although converting to bytecode is extra work it brings two benefits. The first is that the resulting bytecode can be optimised more easily than optimising the syntax tree directly. The second is that the GVMT can produce a bytecode to machine-code compiler automatically.

The Scheme optimiser is a series of passes, which use special interpreters created using GVMTXC (see Section 2.2.1). These passes are defined in the files of the form `/example/XXX.vmc` (except `/example/interpreter.vmc`).

The bytecode to machine-code, or JIT,[1] compiler is generated by the GVMTCC tool from the output of GVMTIC.

1.2 The Tools

The GVMT provides a number of tools, which are described in more detail in Chapter 2. These can be loosely grouped into front-end tools, and back-end tools. Front-end tools are responsible for compiling source code to the GVMT abstract machine code. Back-ends convert this abstract machine code into an executable virtual machine.

1.2.1 The front-end compilers, gvmtc and gvmtic

GVMTC is a C compiler and compiles C files to GSC files. GVMTIC Takes an interpreter specification file and converts into a GSC file. The interpreter specification is in the form of a list of bytecodes; each of which consists of a name, a stack effect comment and a C code (or GSC) body.

Both tools use a modified version of lcc (lcc-gvmt) to do the compilation. See section 2.1.7 for more details.

1.2.2 The code-generators, gvmtas and gvmtcc

The code-generators take GSC as input and create object files or executables.

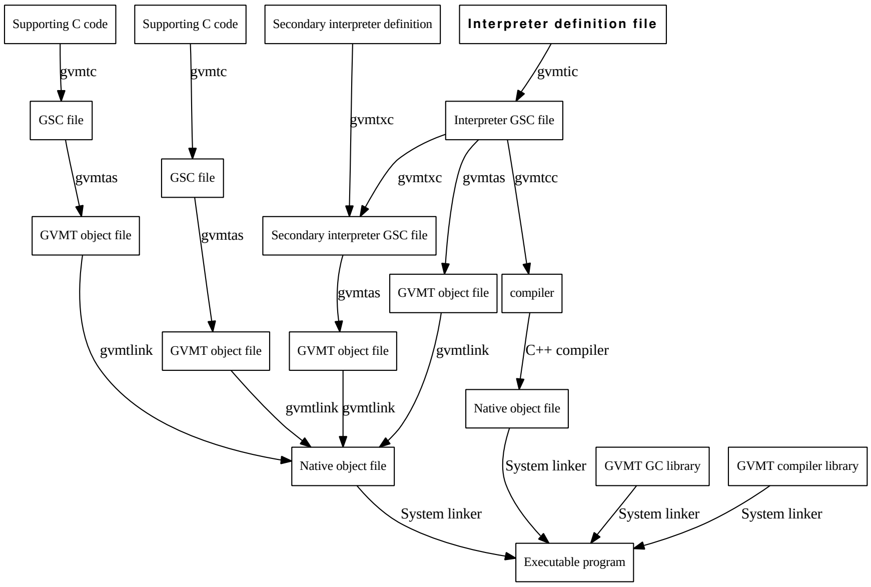
GVMTAS compiles the GSC into the GVMT object format, GSO. It works by converting GSC back into C in such a way as to respect the semantics of the GVMT execution model, on a section-by-section basis. GVMTAS is also responsible for wrapping the bytecodes in an interpreter loop.

GVMTCC takes a GSC file containing a bytecode section and produces a bytecode to machine-code. Currently GVMTCC generates llvm-IR[2] and uses LLVM to generate machine-code at runtime (as a JIT compiler).

[1] JIT is short for Just-In-Time and is the standard term for runtime compilers

[2] See llvm.org for more details

Figure 1.1: Building a Virtual Machine using GVMT



1.2.3 Other tools

GVMTXC takes a GSC file containing a bytecode section representing the main interpreter, plus a secondary file describing a secondary interpreter. It produces an interpreter (as a GSC file) which uses the bytecodes defined for the main interpreter, but with the semantics defined in the secondary file. GVMTXC can be used for creating bytecode verifiers and disassemblers, as well as optimisers and other abstract interpreters.

The bytecode-transformation engines produced by GVMTXC and the compilers produced by GVMTCC can be used independently or combined to form a specialised optimising compiler.

GVMTLINK links together all the GSO files produced by GVMTAS into a single object ready to be linked by the system linker. See Figure 1.1.

1.3 GVMT input files

GVMT takes a number of input files:

- Interpreter description file(s).
- Function definition file(s).

Chapter 2

The tools

GVMT is, as the name suggests, supplies a number of tools. Six of these form the core of the toolkit.

As mentioned in the previous chapter, the GVMT tools can be grouped into front-end and back-end tools. The front-end tools convert source code into code for the GVMT Abstract Machine. Back-end tools convert the abstract representation into real manchine code.

2.1 Core tools

The six core tools are:

- Front end tools
  - GVMTIC
  - GVMTC
  - GVMTXC
- Back end tools
  - GVMTAS
  - GVMTCC
  - GVMTLINK

GVMTIC and GVMTC form the front-end and are used to translate the C-based interpreter description and standard C files, respectively, into GSC format.

GVMTAS converts these GSC files to object files and GVMTLINK links them together. GVMTCC takes the GSC file produced by GVMTIC and produces a compiler from it.

2.1.1 Interpreter description file

An interpreter description file consists of bytecode definitions plus the local variables shared by all bytecodes.

Each bytecode can be defined by its stack effect plus a snippet of C code or as a sequence of other bytecodes and predefined GVMT abstract stack machine codes.

Usually there is one primary interpreter description file that defines the bytecode format. Secondary interpreters (see Sect. 2.2.1 will be guaranteed to use the same bytecode format. However it is possible to have several independent primary interpreters. For example a virtual machine might use one bytecode format for its interpreter and compiler and an entirely different format for its regular-expression engine. Each primary interpreter may have secondary interpreters.

Interpreters are fully re-entrant and thread-safe. The interpreter description file defines not only the behaviour of the interpreter, but the behaviour of the GVMT-generated compiler as well.

Format

The interpreter description file consists of a locals block and any number of bytecode declarations in any order.

Interpreter-local variables

Interpreter-locals variables are declared in a locals section:

```
locals {
        type name;

⋮

}
```

Interpreter-local variables are visible within all bytecode definitions, and can be accessed indirectly via the interpreter frame on the control stack. Interpreter-local variables (and the interpreter frame) have the same lifetime as the invocation of the interpreter.

Bytecode definitions

Bytecodes can be defined either in C or directly in GSC. Both types of definition can be used in the same interpreter.

C-bytecode definitions are in the form

```
name (‘[’ qualifier* ‘]’)?  ‘(’ stack_comment ‘)’ [ ‘=’ <int> ] ‘{’
    Arbitrary C code
‘}’
```

Legal qualifiers are:

private This bytecode is not visible at the interpreter level.

protected This bytecode is acceptable to the interpreter, but not to any verification code. It is only to be used internally as an optimisation. The meaning of ‘protected’ is largely conventional. GVMT will ignore it.

nocomp This bytecode will not appear in bytecode passed to the compiler. If this bytecode is seen by the compiler it will abort. Useful for reducing the size of the compiler.

componly This bytecode will only be passed to the compiler. It will cause the interpreter to abort. Useful for reducing the size of the interpreter.

The `stack_comment` is of the form:

```
type id (‘,’ type, id)* -- type id (‘,’ type, id)*)
```

Where `type` is any legal C type and `id` is any legal C identifier, or a legal C identifier prefixed with `#`. Variables prefixed with `#` are fetched from the instruction stream, not popped from the stack. When referred to in the C code, the `#` prefix is dropped.

The integer at the end of the header line is the actual numerical value of this bytecode. If not supplied, GVMT will allocate bytecode numbers starting from 1.

Compound instructions are in the form:

```
name (‘[’ qualifier* ‘]’)?  ’(’ stack comment ’)’ [ ‘=’ <int> ] ‘:’
   instruction*
‘;’
```

Where `instruction` is a legal GSC or user-defined instruction.

To allow user defined instructions to do anything useful, a large number of builtin instructions are provided. All are private, that is they must be used inside a compound instruction to be visible at the interpreter level.

Instruction stream manipulation operators are provided: `#@` Removes a byte from the instructino stream and pushes it to the evaluation stack. `#N` where N is a one-byte integers pushes N into the front of the instruction stream. `#[N]` Pushes a copy of the Nth item, starting at zero, in the instruction stream into the front of the instruction stream. `#+` or `#-` Adds or subtracts, respectively the first two values from the instruction stream and pushes the result back to the front of the instruction stream.

Appendix B contains a full list of all abstract machine instructions

There are three bytecode names which are treated specially by the GVMT, these are:

- __preamble
- __postamble
- __trace

`__preamble` is executed when the interpreter is called, before any bytecodes are executed.

`__postamble` is executed when the interpreter reaches the end of the bytecodes. Interpreters with a `__postamble` instruction expect a second parameter, the end of the bytecode sequence. This is useful for ‘interpreters’ which analyse code rather than executing it.

`__trace` is executed before the main bytecode definition for every bytecode executed.

Examples

TO DO – Put a couple of examples here. 1 C, 1 vmc.

2.1.2 The interpreter generator gvmtic

GVMTIC comiles an interpreter description file, as described above, into a GSC file.

GVMTIC takes the following options:

-D symbol Define symbol in preprocessor

-I file Add file to list of #include file for lcc sub-process, can be used multiple times

-a Warn about non-ANSI C.

-b bytecode-header-file Specify bytecode header file

-h Print this help and exit

-n name Name of generated interpreter

-o outfile-name Specify output file name

-v Echo sub-processes invoked

-z Do not put gc-safe-points on backward edges

2.1.3 The C compiler gvmtc

GVMTC takes a standard C89 file as its input and outputs a GSC file.

GVMTC takes the following options:

-D symbol Define symbol in preprocessor

-I file Add file to list of #include file for lcc sub-process, can be used multiple times

-L Directory to find lcc executables

-a Warn about non-ANSI C.

-h Print this help and exit

-o outfile-name Specify output file name

-v Echo sub-processes invoked

-x Output dot file of IR (For debugging compiler)

-z Do not put gc-safe-points on backward edges

2.1.4 The GSC assembler gvmtas

GVMTAS takes a GSC file and assembles it to a GVMT object file (.gso is the expected file extension)

GVMTAS takes the following options:

-H dir Look for gvmt internal headers in dir.

-O Optimise. Levels 0 to 3

-T Use token-threading dispatch

-g Debug

-h Print this help and exit

-l Output GSO suitable for library code, no bytecode, root or heap sections allowed

-m memory_manager Memory manager (garbage collector) used

-o outfile-name Specify output file name

-t Turn on tracing (in interpreter)

2.1.5 The compiler generator gvmtcc

GVMTCC Takes a GSC file produced by GVMTIC and produces a compiler. The generated compiler is in C++ and relies on LLVM (http://llvm.org/). It will need to be compiled with the system C++ compiler.

It may be possible to generate compilers using other code-generator backends in the future, but there is currently no plan to do so.

GVMTCC takes the following options:

-h Print this help and exit

-m memory_manager Memory manager (garbage collector) used

-o outfile-name Specify output file name

2.1.6 The linker gvmtlink

GVMTLINK takes any number of GVMT object (.gso) files and links them to form a single native object file. This object file can be linked with the GC object file and compiler object file (if required) using the system linker to form an executable.

GVMTLINK takes the following options:

-h Print this help and exit

-l Output a library (.dll or .so)

-n No-heap, error if heap or roots section exist

-o outfile-name Specify output file name

-v verbose

2.1.7 lcc-gvmt

Both GVMTIC and GVMTC use a modified version of lcc as their C-to-GSC compiler. This compiler uses the standard lcc front-end and a custom back-end. The back-end operates in four passes. The first pass labels the IR forests with the C type for that expression, as much as can be derived without the explicit casts present in the source code. The second pass is a bottom pass which labels the trees with the set of possible GSC types it may take. The third pass is a top-down pass which attempts to find the exact GSC type of each node. The fourth pass emits the GSC.

2.2 Other tools

2.2.1 The bytecode-processor generator gvmtxc

GVMTXC Takes a (partial) interpreter description file and a master GSC file (produced by GVMTIC) and produces a GSC file representing an interpreter. This interpreter is guaranteed to accept the same bytecode format as the master interpreter. For a partial interpreter description file, the generate interpreter skips over the omitted bytecodes.

GVMTXC takes the following options:

-D symbol Define symbol in preprocessor

-I file Add file to list of #include file for lcc sub-process, can be used multiple times

-a Warn about non-ANSI C.

-b bytecode-header-file Specify bytecode header file

-e Explicit: All bytecodes must be explicitly defined

-h Print this help and exit

-n name Name of generated interpreter

-o outfile-name Specify output file name

-v Echo sub-processes invoked

2.2.2 The object layout tool

The object layout tool is an optional tool to help manage heap object layout. It can create C structs, shape vectors and marshalling to GSC files for objects all from a single specification. This tool is designed as an extensible Python script rather than a stand-alone tool, although it can be used as such. When used as a standalone tool it has takes the following options:

-h Print this help and exit

-m Output marshalling code for all kinds

-o outfile-name Specify output file name

-p Define a pattern to use for typedefs

-s Output header with stuct definitions for C code

-t Output header with typedefs for C code

Chapter 3

GVMT Execution Model

The GVMT assists in building a virtual machine(VM) for a high-level language(HLL). Like any virtual machine, the user-defined VM must execute on a lower-level machine (usually a hardware machine). The GVMT defines an abstract stack-based machine (The GVMT Abstract Machine) that has well defined semantics, but is abstract in the sense that it cannot execute any programs. It is designed to both assist in implementing stack-based virtual machine and allow translation to efficient machine-code.

3.1 Components

The GVMT Abstract Machine consists of two parts:

1. A set of executing threads
2. A shared heap

Each thread of execution consists of five components:

1. Temporary registers
2. Control stack
3. Evaluation stack
4. Thread-local variables
5. Exception handling stack

When the GVMT Abstract Machine is started it consists of one thread only.

3.2 Threads

Each thread is independent, the number of concurrently executing threads is limited only by the hardware. Threads share the heap, but have their own stacks, thread-local variables and temporary registers.

3.2.1 Execution Model

The execution model executes abstract HLL virtual machine instructions. Each HLL VM instruction is defined in terms of GSC instructions. GSC instructions can be grouped into three categories:

Evaluation Instructions that take values on the Evaluation stack and place results there, such as addition.

Flow control Instructions that alter the flow of control, both within HLL-VM instructions and between HLL-VM instructions.

Stack manipulation Instructions that explicitly manipulate or describe the status of the evaluation and control stacks.

Appendix B contains a list of all the GSC instructions.

3.2.2 GC Safe points

The GC safe instruction declares that the executing thread can be interupted for garbage collection. In order for this to be safe, the following restrictions must be observed:

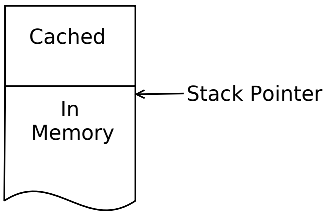
1. No non-reference values are on the execution stack.
2. All heap objects reachable from this thread are in a scannable state.[1]

The front-end tools GVMTC and GVMTIC automatically ensure that condition 1 is true for all C code. The toolkit ensures that all virtual stack items are reachable by the garbage-collector. Condition 2 can be ensure simply by initialising all objects immediately after allocation.

3.2.3 The control stack

The control stack is an opaque structure used to handle procedure calls and returns, exception handling and storing of temporary variable, across calls. Interpreter frames are also allocated on the control stack. Memory can be allocated on the call-stack, but subsequent allocation are not guaranteed to be contiguous. Parameters are also passed to native code using the control stack. The control stack will usually map directly onto the native C stack.

3.2.4 Evaluation stack



[1] Usually this means that the header(class) of any newly created object will have been initialised.

The evaluation stack is where all arithmetic operations takes place. The top part of the evaluation stack may be virtualised, that is stored in registers or other discrete values. The evaluation stack is continuous across calls, and is used for passing parameters. Parameters are pushed to the stack by the caller and popped by the calle. See the GSC instruction `STACK` for more details on how to access the in-memory stack directly.

3.2.5 Thread-local variables

Each thread has thread-locals variables, the number and type of these variables are determined by the developer.

3.3 The heap

The GVMT Abstract Machine heap is a garbage collected heap. The toolkit user needs to provide a function `gvmt_shape` which supplies the garbage collector with information for scanning of objects. See section 5.1.1. Static roots of the heap can either be defined when the VM is built or added at runtime, see section5.2.3.

3.3.1 Shape

During tracing the garbage collector needs to know both the extent of an object and which fields are references to other objects, and which are not. The developer communicates this information to the toolkit via a 'shape' vector (array). This shape is a zero-terminated vector of integers. Each integers represents either N successive references or N successive words of non-references. References are represented by positive numbers, non-references by negative numbers (zero is the terminator). For example the shape [2, -2, 1, 0] represents an object of total extent 5 words, the first 2 and last of which are references. The following C declaration (taken from the example Scheme VM):

```
GVMT_OBJECT(cons) {
    struct type *type;   // Opaque (non-GC) pointer
    GVMT_Object car;     // Reference to heap object
    GVMT_Object cdr;     // Reference to heap object
};
```

defines `cons` objects. These `cons` objects would have a shape of [-1, 2, 0].

3.3.2 Garbage collection

No garbage collection algorithm is specified for the GVMT. However, since the GVMT can support both read and write barriers as well as the declaration of GC safe points, a wide range of collectors should be feasible. Currently both generational and non-generational collectors are available, although only 'stop-the-world' collectors have been implemented. The GVMT garbage collectors also support tagging, which allows non-references to be stored in reference slots by setting one or more 'tag' bits. Read and write barriers are implemented via the RLOAD_R and RSTORE_R instructions respectively.

3.3.3 Exception handling

The GVMT exception handling model uses an explicit stack, rather than tables. These means that using exceptions has a runtime cost even when the exceptions are not used as a means of flow control transfer, but that exception handling itself is much faster than for table-based approaches. The GVMT exception handling mechanism is similar to the C setjump-longjump mechanism. The three GSC instructions involved are (See section 5.2.4 for the matching intrinsics)

PROTECT pushes a handler onto the exception stack. This handler holds the following information: The current execution point(immediately after the PROTECT instruction), the stack depth, and the current control stack frame. Finally it pushes NULL onto the evaluation stack.

UNPROTECT pops a handler from the exception stack.

RAISE Pops the value from the TOS and stores it. The exception handler on top of the exception stack is examined and the thread execution state (evaluation and control stack pointers) restored to the values held in the handler. The stored value is pushed onto the evaluation stack. Finally, execution resumes from the location stored in the exceptino handler.

It is an error (resulting in a probable crash) to leave an exception handler on the exception stack after the procedure in which it was pushed has returned. There is no concept of catching only some sorts of exceptions in the GVMT. All exceptions are caught; exceptions that should be handled elsewhere must be explicitly reraised.

Exceptions work as expected in interpreted (and compiled) code. The interpreter instruction pointer is stored along with the machine instruction pointer.

3.4 Concurrency

The GVMT is concurrency friendly, all generated code and library code is thread-safe. The GVMT does not provide a thread library, so the developer will have to use the underlying operating system library. Concurrency is at the thread level, the GVMT is unaware of processes.

In order that the toolkit can operate correctly, it must be informed of the creation and destruction of threads. The toolkit provides functions for this purpose, see Section 5.2.5 for more details. The developer should also be aware that heap allocation may block, waiting for a collection, so locks should not be held across an allocation.

The GVMT also provides fast locks for synchronisation in the case where contention is expected to be rare. This functionality is provided by the LOCK and UNLOCK abstract machine instructions.

The intrinsic functions `gvmt_lock(gvmt_lock_t *lock)` and `gvmt_unlock(gvmt_lock_t *lock)` are translated to the LOCK and UNLOCK instructions, respectively.

Warning: GVMT locks are not automatically unlocked by the RAISE instruction. The developer must ensure that either a RAISE cannot occur between a LOCK and UNLOCK, or that the LOCK UNLOCK pair are PROTECTed. A safe LOCK/UNLOCK sequence is:

```
gvmt_lock_t lock = GVMT_LOCK_INITIALIZER;
GVMT_Object ex;
gvmt_lock(&lock)
GVMT_BEGIN_TRY(ex)

⋮

gvmt_unlock(&lock)
GVMT_CATCH
gvmt_unlock(&lock)
gvmt_raise(ex)
GVMT_END_TRY
```

3.4.1 Memory model

As all threads share the same heap, they may access heap objects concurrently. The following guarantees are made about the state of the heap and roots seen from one thread when modified by another thread:

All writes to pointers(P), references(R) and integers(I) not larger than a pointer are atomic.

The order in which changes to memory appear to one thread are the same as for another. Ie, thread 1 writes to A, then writes to B. Thread 2 will not see a change in B before it sees a change in A.

In between synchronisation points, the delay between a write being made by one thread and a write being seen by another thread is indefinite.

3.5 GVMT Stack Code

GVMT Stack Code (GSC) is the instruction set of the GVMT Abstract Machine. GSC should be viewed as a sort of abstract assembly language. It is possible to define a virtual machine entirely in C, as the front-end tools, GVMTIC and GVMTC, compile C into GSC. It is useful, however to have some awareness of GVMT Stack Code when using the GVMT. To get a better idea of how GSC is used, have a look at the output of GVMTCand GVMTICin the `/example` directory.

Appendix B contains a full list of GSC instructions.

Chapter 4

Building a VM using the Toolkit

4.1 Before you start

Before starting a virtual machine, you will need to generate some bytecode programs to execute. If you are targetting a pre-existing VM format, this is straightforward. If you are creating a new language you will need a parser. GVMTIC can produce a C header file containing definitions for all the opcodes: the C macro for the opcode for instruction 'xxx' is `GVMT_OPCODE(xxx)`.

4.2 Defining the components

For GVMT to build a VM, you will need to define:

- An interpreter. See section 2.1.2
- Any other support code that your interpreter calls. See section 2.1.3

You will also need to define a small number of functions required by the GVMT. The header file for these can be found in /include/user.h and sample implementations can be found in `/example/native.s` and `/example/support.c` It may also be useful to specify:

- The object layout for heap objects. See section 2.2.2

Once these are defined, the VM is built as follows, see figure 1.1: The `.gsc` files for the interpreter and other support code is created by running GVMTIC and GVMTC respectively. These `.gsc` files are converted to `.gso` with GVMTAS. All the `.gso` files are then linked together using GVMTLINK to form as single object file, which can be linked with any native object files to form an executable using the standard linker.

4.3 Debugging

Although the toolkit maintains symbolic information from the source to the executable, some variables may get renamed. The signatures of functions may appear different in the debugging, although their names should be unaltered. The execution stack can be accessed via the stack pointer, the name of which is `gvmt_sp`.

4.4 Adding a compiler

It is probably best to have a working interpreter before adding the compiler. To generate the compiler simply run GVMTCC on the output file from GVMTIC. The resulting C++ file should be compiled and linked using the system C++ compiler. See section 2.1.5 for more details.

Chapter 5

User interface

To create a complete VM, the various components need to interact and to do so correctly.

So this can happen the VM developer is required to implement a number of functions to assist the GVMT. The GVMT provides an API for interacting with generated and library components.

Examining the files in the /example subdirectory may help to make things clearer.

5.1 User provided code

The file /include/gvmt/user.h lists all the functions which the developer needs to supply, with the exception of the `gvmt_shape` function which must be compiled natively, not by GVMTC.

5.1.1 Garbage collection

In order for the tooolkit to provide precise garbage collection, the layout of all heap objects must be known. The developer must provide two functions and one integer value to allow GVMT to perform *precise* garbage collection.

int GVMT_MAX_SHAPE_SIZE The maximum number of entries required to represent any shape (including terminating zero).

long* gvmt_shape(GVMT_Object object, int* buf) This function returns a vector of integers, representing the shape of the object, as defined in section 3.3.1. The int array buf can be used to store the shape if required. The buffer, `buf` is guaranteed to include at least GVMT_MAX_SHAPE_SIZE entries.

long gvmt_length(GVMT_Object obj) This function returns the length of the object in bytes.

void user_finalize_object(GVMT_Object o) This function may be called by the garbage collector for any object that is declared as requiring finalization and is unreachable. Exceptions raised in this function, or any callee, terminate finalisation, but do not propogate. The function is called by the finalizer thread and will be called asynchronously. This function may be called on any or all object declared as finalizable, in any order.

5.1.2 The Marshalling Interface

In order to support marshalling the user must provide a function `gvmt_write_func get_marshaller_for_object(GVMT_Object object)` to give a write function for each object. The toolkit can create a write function for any object declaration, but it is unable to determine which function should be applied to which object at runtime.

To assist marshalling to GSC code, the user should also provide the `void gvmt_readable_name(GVMT_Object object, char *buffer)` function to provide meaningful names for objects. This function should store an ASCII string into the provided buffer. The buffer is guaranteed to be of at least GVMT_MAX_OBJECT_NAME_LENGTH bytes in length

5.1.3 Debugging

The GVMT can insert calls to a trace function at the start of each bytecode. Since GVMT knows nothing about the semantics of the VM, the developer must provide the actual trace function. A very simple version is provided in the example.

5.2 GVMT provided functions

GVMT provides a wide range of functions to interact with the underlying abstract machine, as well as a number of intrinsic functions for code written in C.

5.2.1 The evaluation stack

The evaluation stack can be examined and modified in C code, using intinsics:

- `gvmt_stack_top()` returns a pointer to the top-of-stack.
- `gvmt_insert(size_t n)` inserts n NULLs onto the stack and returns a pointer to them.
- `GVMT_PUSH(x)` can be used to push individual values on to the stack.
- `gvmt_drop(size_t n)` discards n items from the top of the stack.

Care should be taken with these functions as they manipulate the evaluation stack, which is used to evaluate expression. It is best to keep any C statement using these intrinsics as simple as possible.

5.2.2 The call stack

The call stack is generally opaque, except for interpreter frames, which can be accessed using the following intinsics:

- gvmt_current_frame() returns a opaque pointer to the currently executing interpreter frame.
- gvmt_caller_frame(void* frame) takes a pointer to an interpreter frame and returns a pointer to its caller.
- gvmt_frame_index(char* name) returns an index for a named local. This index can be used to read the local variables of a given frame.
- gvmt_frame_item_r(void* frame, unsigned index) returns the reference local variable corresponding to the given index.
- gvmt_frame_item_p(void* frame, unsigned index) returns the pointer local variable corresponding to the given index.
- gvmt_frame_item_i(void* frame, unsigned index) returns the integer local variable corresponding to the given index.

5.2.3 The garbage collector

The following intrinsic allocates object (the GSC code is GC_MALLOC).

- GVMT_Object gvmt_malloc(size_t s) Allocates a new object. This object should be initialised immediately, and *must* be initialised before the next GC-SAFE point.

The garbage collector can automatically find all objects from both stacks and all global variables. Sometimes it is necessary to have some control over the garbage collector. GVMT provides the following functions to interact with the garbage collector.

- gvmt_gc_finalizable(GVMT_Object obj) Declares that obj will need finalization.
- void* gvmt_gc_weak_reference(void) Returns a weak reference to obj.
- GVMT_Object gvmt_gc_read_weak_reference(void* w) Reads a weak reference, the reference returned will either be the last value written to this weak-reference or NULL.
- gvmt_gc_read_weak_reference(void* w, GVMT_Object o) Writes to a weak reference
- void gvmt_gc_free_weak_reference(void* w) Frees a weak reference if no longer required.
- void* gvmt_gc_add_root(void) Returns a new root to the heap, initialised to obj.
- GVMT_Object gvmt_gc_read_root(void* root) Returns the value referred to by the root.
- void gvmt_gc_write_root(void* root, GVMT_Object obj) Writes a new object to a root.

- `void gvmt_gc_free_root(void* root)` Deletes this root, the object referred to may now be garbage collected.

- `void* gvmt_gc_pin(GVMT_Object obj)` Pins the object referred to by `obj`. The garbage collector will not move it. Be aware that collection and pinning are independent. The grabge collector may collect pinned objects. To pass an object to native code, or to use an internal pointer will require both pinning the object *and* retaining a refeerence to it.

- `gvmt_gc_unpin(GVMT_Object obj)` Unpins the object, which may now by moved by the garbage collector. It is not necessary to unpin objects in order for them to be garbage collected.

5.2.4 Exception handling

Three intrinsics are provided:

- `GVMT_Object gvmt_protect(void)` Intrinsic for the PROTECT instruction.

- `void gvmt_unprotect(void)` Intrinsic for the UNPROTECT instruction.

- `void gvmt_raise(GVMT_Object ex)` Intrinsic for the RAISE instruction.

Rather than use `GVMT_Object gvmt_protect()` and `void gvmt_unprotect()` directly developers should use the macros: `GVMT_TRY`, `GVMT_CATCH` and `GVMT_END_TRY` as follows:

```
GVMT_TRY(exception_variable)
    code which may raise exception
GVMT_CATCH
    code to execute if an exception is raised
GVMT_END_TRY
```

5.2.5 Threading

Since the GVMT relies on the operating system to provide threading support, it must be informed when a new thread is created and used. To start running GVMT code in a new thread, call:

```
int gvmt_enter(uintptr_t stack_space, gvmt_func_ptr func, int pcount, ...);
```

Where:

- `stack_space` is the amount of items required on the evaluation stack,

- `gvmt_func_ptr` is the GSC function to execute, and

- `pcount` is the number of parameters following.

If reentering GVMT from native code that was is running in an already initialised thread, then use

```
int gvmt_reenter(gvmt_func_ptr func, int pcount, ...);
```

to avoid reinitialising the running thread.

When a thread has finished call `gvmt_finished(void)`.

Appendix A

Installing the GVMT

Currently the GVMT has only been tested on the x86-linux platform, but it should work on any posix-compliant x86 platform. GVMT can be installed in the following steps:

1. Download GVMT from `http://code.google.com/p/gvmt/`
2. Unpack into a clean directory, $(GVMT)
3. Install llvm (version2.5) from `http://llvm.org/releases/download.html#2.5`
4. Type `make`

   The build process will fetch lcc using wget. If you do not have wget installed or wget fails then you will have to download lcc from `http://sites.google.com/site/lccretargetablecompiler/downloads/4.2.tar.gz` and save it as $(GVMT)/lcc.tar.gz

5. Type `sudo make install` (You will need write privileges to /usr/local in order to install the tools)

If it doesn't work, then email me: marks@dcs.gla.ac.uk

Appendix B

GSC Instruction set

Introduction

This appendix lists all 293 instructions of the GSC instruction set. The GSC instruction set is not as large as it first appears. Many of these are multiple versions of the form OP_X where X can be any or all of the twelve different types. These types are I1, I2, I4, I8, U1, U2, U4, U8, F4, F8, P, R.

IX, UX and FX refer to a signed integer, unsigned integer and floating point real of size (in bytes) X. P is a pointer and R is a reference. P pointers cannot point into the GC heap. R references are pointers that can *only* point into the GC heap.

TOS is an abbreviation for top-of-stack and NOS is an abbreviation for next-on-stack.

Each instruction is listed below in the form:

Name (inputs ⇒ outputs)

Instruction stream effect
Description of the instruction

#+ (— ⇒ —)

2 operand bytes. Pushes 1 byte to instruction stream.
Fetches the first two values in the instruction stream, adds them and pushes the result back to the stream.

#- (— ⇒ —)

2 operand bytes. Pushes 1 byte to instruction stream.
Fetches the first two values in the instruction stream, subtracts them and pushes the result back to the stream.

#N (— ⇒ —)

No operand bytes. Pushes 1 byte to instruction stream.
Push 1 byte value to the front of the instruction stream.

#2@ (— ⇒ operand)

2 operand bytes.
Fetches the next 2 byte(s) from the instruction stream and pushes it onto the evaluation stack.

#4@ (— ⇒ operand)

4 operand bytes.
Fetches the next 4 byte(s) from the instruction stream and pushes it onto the evaluation stack.

#@ (— ⇒ operand)

1 operand byte.
Fetches the next 1 byte(s) from the instruction stream and pushes it onto the evaluation stack.

#[N] (— ⇒ —)

No operand bytes. Pushes 1 byte to instruction stream.
Only valid in bytecode context. Peeks into the instruction stream and pushes the N^{th} byte in the stream to the front of the instruction stream.

ADDR(X) (— ⇒ address)

Address of the global variable X

ADD_F4 (op1, op2 ⇒ value)

Binary operation: 32 bit floating point add.

ADD_F8 (op1, op2 ⇒ value)

Binary operation: 64 bit floating point add.

ADD_I4 (op1, op2 ⇒ value)

Binary operation: 32 bit signed integer add.

ADD_I8 (op1, op2 ⇒ value)

Binary operation: 64 bit signed integer add.

ADD_P (op1, op2 ⇒ value)

Binary operation: pointer add.

ADD_U4 (op1, op2 ⇒ value)

Binary operation: 32 bit unsigned integer add.

ADD_U8 (op1, op2 ⇒ value)

Binary operation: 64 bit unsigned integer add.

ALLOCA_F4 (N ⇒ ptr)

Allocates space for N 32 bit floating points in the current control stack frame, leaving pointer to allocated space in TOS. All memory allocated after a PROTECT is invalidated immediately by a RAISE, but not necessarily immediately reclaimed. All memory allocated is invalidated and reclaimed by a RETURN instruction.

ALLOCA_F8 (N ⇒ ptr)

Allocates space for N 64 bit floating points in the current control stack frame, leaving pointer to allocated space in TOS. All memory allocated after a PROTECT is invalidated immediately by a RAISE, but not necessarily immediately reclaimed. All memory allocated is invalidated and reclaimed by a RETURN instruction.

ALLOCA_I1 (N ⇒ ptr)

Allocates space for N 8 bit signed integers in the current control stack frame, leaving pointer to allocated space in TOS. All memory allocated after a PROTECT is invalidated immediately by a RAISE, but not necessarily immediately reclaimed. All memory allocated is invalidated and reclaimed by a RETURN instruction.

ALLOCA_I2 (N ⇒ ptr)

Allocates space for N 16 bit signed integers in the current control stack frame, leaving pointer to allocated space in TOS. All memory allocated after a PROTECT is invalidated immediately by a RAISE, but not necessarily immediately reclaimed. All memory allocated is invalidated and reclaimed by a RETURN instruction.

ALLOCA_I4 (N ⇒ ptr)

Allocates space for N 32 bit signed integers in the current control stack frame, leaving pointer to allocated space in TOS. All memory allocated after a PROTECT is invalidated immediately by a RAISE, but not necessarily immediately reclaimed. All memory allocated is invalidated and reclaimed by a RETURN instruction.

ALLOCA_I8 (N ⇒ ptr)

Allocates space for N 64 bit signed integers in the current control stack frame, leaving pointer to allocated space in TOS. All memory allocated after a PROTECT is invalidated immediately by a RAISE, but not necessarily immediately reclaimed. All memory allocated is invalidated and reclaimed by a RETURN instruction.

ALLOCA_P (N ⇒ ptr)

Allocates space for N pointers in the current control stack frame, leaving pointer to allocated space in TOS. All memory allocated after a PROTECT is invalidated immediately by a RAISE, but not necessarily immediately reclaimed. All memory allocated is invalidated and reclaimed by a RETURN instruction.

ALLOCA_R (N ⇒ ptr)

Allocates space for N references in the current control stack frame, leaving pointer to allocated space in TOS. All memory allocated after a PROTECT is invalidated immediately by a RAISE, but not necessarily immediately reclaimed. All memory allocated is invalidated and reclaimed by a RETURN instruction.

ALLOCA_U1 (N ⇒ ptr)

Allocates space for N 8 bit unsigned integers in the current control stack frame, leaving pointer to allocated space in TOS. All memory allocated after a PROTECT is invalidated immediately by a RAISE, but not necessarily immediately reclaimed. All memory allocated is invalidated and reclaimed by a RETURN instruction.

ALLOCA_U2 (N ⇒ ptr)

Allocates space for N 16 bit unsigned integers in the current control stack frame, leaving pointer to allocated space in TOS. All memory allocated after a PROTECT is invalidated immediately by a RAISE, but not necessarily immediately reclaimed. All memory allocated is invalidated and reclaimed by a RETURN instruction.

ALLOCA_U4 (N ⇒ ptr)

Allocates space for N 32 bit unsigned integers in the current control stack frame, leaving pointer to allocated space in TOS. All memory allocated after a PROTECT is invalidated immediately by a RAISE, but not necessarily immediately reclaimed. All memory allocated is invalidated and reclaimed by a RETURN instruction.

ALLOCA_U8 (N ⇒ ptr)

Allocates space for N 64 bit unsigned integers in the current control stack frame, leaving pointer to allocated space in TOS. All memory allocated after a PROTECT is invalidated immediately by a RAISE, but not necessarily immediately reclaimed. All memory allocated is invalidated and reclaimed by a RETURN instruction.

AND_I4 (op1, op2 ⇒ value)

Binary operation: 32 bit signed integer bitwise and.

AND_I8 (op1, op2 ⇒ value)

Binary operation: 64 bit signed integer bitwise and.

AND_U4 (op1, op2 ⇒ value)

Binary operation: 32 bit unsigned integer bitwise and.

AND_U8 (op1, op2 ⇒ value)

Binary operation: 64 bit unsigned integer bitwise and.

BRANCH_F(N) (cond ⇒ —)

Branch if TOS is False to Target(N).

BRANCH_T(N) (cond ⇒ —)

Branch if TOS is True to Target(N).

CALL_F4 (— ⇒ value)

Calls the function whose address is TOS. Pushes the return value which must be of type float32.

CALL_F8 (— ⇒ value)

Calls the function whose address is TOS. Pushes the return value which must be of type float64.

CALL_I4 (— ⇒ value)

Calls the function whose address is TOS. Pushes the return value which must be of type int32.

CALL_I8 (— ⇒ value)

Calls the function whose address is TOS. Pushes the return value which must be of type int64.

CALL_P (— ⇒ value)

Calls the function whose address is TOS. Pushes the return value which must be of type ptr.

CALL_R (— ⇒ value)

Calls the function whose address is TOS. Pushes the return value which must be of type R.

CALL_U4 (— ⇒ value)

Calls the function whose address is TOS. Pushes the return value which must be of type uint32.

CALL_U8 (— ⇒ value)

Calls the function whose address is TOS. Pushes the return value which must be of type uint64.

CALL_V (— ⇒ value)

Calls the function whose address is TOS. Pushes the return value which must be of type void.

D2F (val ⇒ result)

Converts 64 bit floating point to 32 bit floating point.

D2I (val ⇒ result)

Converts 64 bit floating point to 32 bit signed integer.

D2L (val ⇒ result)

Converts 64 bit floating point to 64 bit signed integer.

DIV_F4 (op1, op2 ⇒ value)

Binary operation: 32 bit floating point divide.

DIV_F8 (op1, op2 ⇒ value)

Binary operation: 64 bit floating point divide.

DIV_I4 (op1, op2 ⇒ value)

Binary operation: 32 bit signed integer divide.

DIV_I8 (op1, op2 ⇒ value)

Binary operation: 64 bit signed integer divide.

DIV_U4 (op1, op2 ⇒ value)

Binary operation: 32 bit unsigned integer divide.

DIV_U8 (op1, op2 ⇒ value)

Binary operation: 64 bit unsigned integer divide.

DROP (top ⇒ —)

Drops the top value from the stack.

DROP_N (count ⇒ —)

1 operand byte.
Pops count off the stack. Drops count values from the stack at offset fetched from stream.

EQ_F4 (op1, op2 ⇒ value)

Comparison operation: 32 bit floating point equals.

EQ_F8 (op1, op2 ⇒ value)

Comparison operation: 64 bit floating point equals.

EQ_I4 (op1, op2 ⇒ value)

Comparison operation: 32 bit signed integer equals.

EQ_I8 (op1, op2 ⇒ value)

Comparison operation: 64 bit signed integer equals.

EQ_P (op1, op2 ⇒ value)

Comparison operation: pointer equals.

EQ_R (op1, op2 ⇒ value)

Comparison operation: reference equals.

EQ_U4 (op1, op2 ⇒ value)

Comparison operation: 32 bit unsigned integer equals.

EQ_U8 (op1, op2 ⇒ value)

Comparison operation: 64 bit unsigned integer equals.

EXT_I1 (value ⇒ extended)

Extends I1 value to full word.

EXT_I2 (value ⇒ extended)

Extends I2 value to full word.

EXT_I4 (value ⇒ extended)

Extends I4 value to full word.

EXT_U1 (value ⇒ extended)

Extends U1 value to full word.

EXT_U2 (value ⇒ extended)

Extends U2 value to full word.

EXT_U4 (value ⇒ extended)

Extends U4 value to full word.

F2D (val ⇒ result)

Converts 32 bit floating point to 64 bit floating point.

F2I (val ⇒ result)

Converts 32 bit floating point to 32 bit signed integer.

F2L (val ⇒ result)

Converts 32 bit floating point to 64 bit signed integer.

FAR_JUMP (ip ⇒ —)

Continue interpretation, with the current abstract machine state, at the IP popped from the stack. If this instruction is executed in compiled code, execution leaves the compiler and continues in the *interpreter*. In order to ensure jumps are compiled use JUMP instead. However, FAR_JUMP can be used as a means of exiting compiled code and resuming interpretation.

FILE(X) (— ⇒ —)

Declares the source file for this code. Like #FILE in C.

FLUSH (— ⇒ —)

Flushes evaluation stack to memory.

FRAME (— ⇒ frame)

Pushes an opaque pointer representing the current interpreter frame, if in the interpreter, or the most recently executed interpreter frame, if not in the interpreter, to TOS.

FULLY_INITIALIZED (object ⇒ —)

Declare TOS object to be fully-initialised. This allows optimisations to be made by the toolkit. Drops TOS as a side effect.

GC_MALLOC (N ⇒ ptr)

Allocates N bytes in the heap leaving pointer to allocated space in TOS. GC pass may replace with a faster inline version. Defaults to GC_MALLOC_CALL.

GC_MALLOC_CALL (N ⇒ ptr)

Allocates N bytes, via a call to the GC collector. Generally users should use GC_MALLOC and allow the toolkit to substitute appropriate inline code.

GC_MALLOC_FAST (N ⇒ ptr)

Fast allocates N bytes, ptr is 0 if cannot allocate fast. Generally users should use GC_MALLOC and allow the toolkit to substitute appropriate inline code.

GC_SAFE (— ⇒ —)

Declares this point to be a safe point for Garbage Collection to occur at. GC pass should replace with a custom version. Defaults to GC_SAFE_CALL.

GC_SAFE_CALL (— ⇒ —)

Calls GC to inform it that calling thread is safe for Garbage Collection. Generally users should use GC_SAFE and allow the toolkit to substitute appropriate inline code.

GE_F4 (op1, op2 ⇒ value)

Comparison operation: 32 bit floating point greater than or equals.

GE_F8 (op1, op2 ⇒ value)

Comparison operation: 64 bit floating point greater than or equals.

GE_I4 (op1, op2 ⇒ value)

Comparison operation: 32 bit signed integer greater than or equals.

GE_I8 (op1, op2 ⇒ value)

Comparison operation: 64 bit signed integer greater than or equals.

GE_P (op1, op2 ⇒ value)

Comparison operation: pointer greater than or equals.

GE_U4 (op1, op2 ⇒ value)

Comparison operation: 32 bit unsigned integer greater than or equals.

GE_U8 (op1, op2 ⇒ value)

Comparison operation: 64 bit unsigned integer greater than or equals.

GT_F4 (op1, op2 ⇒ value)

Comparison operation: 32 bit floating point greater than.

GT_F8 (op1, op2 ⇒ value)

Comparison operation: 64 bit floating point greater than.

GT_I4 (op1, op2 ⇒ value)

Comparison operation: 32 bit signed integer greater than.

GT_I8 (op1, op2 ⇒ value)

Comparison operation: 64 bit signed integer greater than.

GT_P (op1, op2 ⇒ value)

Comparison operation: pointer greater than.

GT_U4 (op1, op2 ⇒ value)

Comparison operation: 32 bit unsigned integer greater than.

GT_U8 (op1, op2 ⇒ value)

Comparison operation: 64 bit unsigned integer greater than.

HOP(N) (— ⇒ —)

Jump (unconditionally) to TARGET(N).

I2D (val ⇒ result)

Converts 32 bit signed integer to 64 bit floating point.

I2F (val ⇒ result)

Converts 32 bit signed integer to 32 bit floating point.

INSERT (count ⇒ address)

1 operand byte.
Pops count off the stack. Inserts count NULLs into the stack at offset fetched from the instruction stream. Ensures that all inserted values are flushed to memory. Pushes the address of first inserted slot to the stack.

INV_I4 (op1 ⇒ value)

Unary operation: 32 bit signed integer bitwise invert.

INV_I8 (op1 ⇒ value)

Unary operation: 64 bit signed integer bitwise invert.

INV_U4 (op1 ⇒ value)

Unary operation: 32 bit unsigned integer bitwise invert.

INV_U8 (op1 ⇒ value)

Unary operation: 64 bit unsigned integer bitwise invert.

IP (— ⇒ instruction_pointer)

Pushes the (interpreter) instruction pointer to TOS.

JUMP (— ⇒ —)

2 operand bytes.
Only valid in bytecode context. Performs VM jump. Jumps by N bytes, where N is the next two-byte value in the instruction stream.

L2D (val ⇒ result)

Converts 64 bit signed integer to 64 bit floating point.

L2F (val ⇒ result)

Converts 64 bit signed integer to 32 bit floating point.

L2I (val ⇒ result)

Converts 64 bit signed integer to 32 bit signed integer.

LADDR(X) (— ⇒ X)

Pushes the address of the local variable 'X' to TOS.

LE_F4 (op1, op2 ⇒ value)

Comparison operation: 32 bit floating point less than or equals.

LE_F8 (op1, op2 ⇒ value)

Comparison operation: 64 bit floating point less than or equals.

LE_I4 (op1, op2 ⇒ value)

Comparison operation: 32 bit signed integer less than or equals.

LE_I8 (op1, op2 ⇒ value)

Comparison operation: 64 bit signed integer less than or equals.

LE_P (op1, op2 ⇒ value)

Comparison operation: pointer less than or equals.

LE_U4 (op1, op2 ⇒ value)

Comparison operation: 32 bit unsigned integer less than or equals.

LE_U8 (op1, op2 ⇒ value)

Comparison operation: 64 bit unsigned integer less than or equals.

LINE(N) (— ⇒ —)

Set the source code line number of the source code. Like #LINE in C.

LOCK (lock ⇒ —)

Locks the fast-lock popped from TOS

LOCK_INTERNAL (offset, object ⇒ —)

Locks the fast-lock in object at TOS at offset NOS. Pops both object and offset from stack.

LSH_I4 (op1, op2 ⇒ value)

Binary operation: 32 bit signed integer left shift.

LSH_I8 (op1, op2 ⇒ value)

Binary operation: 64 bit signed integer left shift.

LSH_U4 (op1, op2 ⇒ value)

Binary operation: 32 bit unsigned integer left shift.

LSH_U8 (op1, op2 ⇒ value)

Binary operation: 64 bit unsigned integer left shift.

LT_F4 (op1, op2 ⇒ value)

Comparison operation: 32 bit floating point less than.

LT_F8 (op1, op2 ⇒ value)

Comparison operation: 64 bit floating point less than.

LT_I4 (op1, op2 ⇒ value)

Comparison operation: 32 bit signed integer less than.

LT_I8 (op1, op2 ⇒ value)

Comparison operation: 64 bit signed integer less than.

LT_P (op1, op2 ⇒ value)

Comparison operation: pointer less than.

LT_U4 (op1, op2 ⇒ value)

Comparison operation: 32 bit unsigned integer less than.

LT_U8 (op1, op2 ⇒ value)

Comparison operation: 64 bit unsigned integer less than.

MOD_I4 (op1, op2 ⇒ value)

Binary operation: 32 bit signed integer modulo.

MOD_I8 (op1, op2 ⇒ value)

Binary operation: 64 bit signed integer modulo.

MOD_U4 (op1, op2 ⇒ value)

Binary operation: 32 bit unsigned integer modulo.

MOD_U8 (op1, op2 ⇒ value)

Binary operation: 64 bit unsigned integer modulo.

MUL_F4 (op1, op2 ⇒ value)

Binary operation: 32 bit floating point multiply.

MUL_F8 (op1, op2 ⇒ value)

Binary operation: 64 bit floating point multiply.

MUL_I4 (op1, op2 ⇒ value)

Binary operation: 32 bit signed integer multiply.

MUL_I8 (op1, op2 ⇒ value)

Binary operation: 64 bit signed integer multiply.

MUL_U4 (op1, op2 ⇒ value)

Binary operation: 32 bit unsigned integer multiply.

MUL_U8 (op1, op2 ⇒ value)

Binary operation: 64 bit unsigned integer multiply.

NAME(N,X) (— ⇒ —)

Name the N$^{\text{th}}$ temporary variable, for debugging purposes.

NARG_F4 (val ⇒ —)

Native argument of type 32 bit floating point.TOS is pushed to the native argument stack.

NARG_F8 (val ⇒ —)

Native argument of type 64 bit floating point.TOS is pushed to the native argument stack.

NARG_I4 (val ⇒ —)

Native argument of type 32 bit signed integer.TOS is pushed to the native argument stack.

NARG_I8 (val ⇒ —)

Native argument of type 64 bit signed integer.TOS is pushed to the native argument stack.

NARG_P (val ⇒ —)

Native argument of type pointer.TOS is pushed to the native argument stack.

NARG_U4 (val ⇒ —)

Native argument of type 32 bit unsigned integer.TOS is pushed to the native argument stack.

NARG_U8 (val ⇒ —)

Native argument of type 64 bit unsigned integer.TOS is pushed to the native argument stack.

NEG_F4 (op1 ⇒ value)

Unary operation: 32 bit floating point negate.

NEG_F8 (op1 ⇒ value)

Unary operation: 64 bit floating point negate.

NEG_I4 (op1 ⇒ value)

Unary operation: 32 bit signed integer negate.

NEG_I8 (op1 ⇒ value)

Unary operation: 64 bit signed integer negate.

NEXT_IP (— ⇒ instruction_pointer)

Pushes the (interpreter) instruction pointer for the *next* instruction to TOS.

NE_F4 (op1, op2 ⇒ value)

Comparison operation: 32 bit floating point not equals.

NE_F8 (op1, op2 ⇒ value)

Comparison operation: 64 bit floating point not equals.

NE_I4 (op1, op2 ⇒ value)

Comparison operation: 32 bit signed integer not equals.

NE_I8 (op1, op2 ⇒ value)

Comparison operation: 64 bit signed integer not equals.

NE_P (op1, op2 ⇒ value)

Comparison operation: pointer not equals.

NE_R (op1, op2 ⇒ value)

Comparison operation: reference not equals.

NE_U4 (op1, op2 ⇒ value)

Comparison operation: 32 bit unsigned integer not equals.

NE_U8 (op1, op2 ⇒ value)

Comparison operation: 64 bit unsigned integer not equals.

N_CALL_F4(N) (— ⇒ value)

Calls the function whose address is TOS. Uses the native calling convention for this platform with 0 parameters which are popped from the native argument stack. Pushes the return value which must be of type float32.

N_CALL_F8(N) (— ⇒ value)

Calls the function whose address is TOS. Uses the native calling convention for this platform with 0 parameters which are popped from the native argument stack. Pushes the return value which must be of type float64.

N_CALL_I4(N) (— ⇒ value)

Calls the function whose address is TOS. Uses the native calling convention for this platform with 0 parameters which are popped from the native argument stack. Pushes the return value which must be of type int32.

N_CALL_I8(N) (— ⇒ value)

Calls the function whose address is TOS. Uses the native calling convention for this platform with 0 parameters which are popped from the native argument stack. Pushes the return value which must be of type int64.

N_CALL_NO_GC_F4(N) (— ⇒ value)

As N_CALL_F4(N). Garbage collection is suspended during this call. Only use the NO_GC variant for calls which cannot block. If unsure use N_CALL.

N_CALL_NO_GC_F8(N) (— ⇒ value)

As N_CALL_F8(N). Garbage collection is suspended during this call. Only use the NO_GC variant for calls which cannot block. If unsure use N_CALL.

N_CALL_NO_GC_I4(N) (— ⇒ value)

As N_CALL_I4(N). Garbage collection is suspended during this call. Only use the NO_GC variant for calls which cannot block. If unsure use N_CALL.

N_CALL_NO_GC_I8(N) (— ⇒ value)

As N_CALL_I8(N). Garbage collection is suspended during this call. Only use the NO_GC variant for calls which cannot block. If unsure use N_CALL.

N_CALL_NO_GC_P(N) (— ⇒ value)

As N_CALL_P(N). Garbage collection is suspended during this call. Only use the NO_GC variant for calls which cannot block. If unsure use N_CALL.

N_CALL_NO_GC_R(N) (— ⇒ value)

As N_CALL_R(N). Garbage collection is suspended during this call. Only use the NO_GC variant for calls which cannot block. If unsure use N_CALL.

N_CALL_NO_GC_U4(N) (— ⇒ value)

As N_CALL_U4(N). Garbage collection is suspended during this call. Only use the NO_GC variant for calls which cannot block. If unsure use N_CALL.

N_CALL_NO_GC_U8(N) (— ⇒ value)

As N_CALL_U8(N). Garbage collection is suspended during this call. Only use the NO_GC variant for calls which cannot block. If unsure use N_CALL.

N_CALL_NO_GC_V(N) (— ⇒ value)

As N_CALL_V(N). Garbage collection is suspended during this call. Only use the NO_GC variant for calls which cannot block. If unsure use N_CALL.

N_CALL_P(N) (— ⇒ value)

Calls the function whose address is TOS. Uses the native calling convention for this platform with 0 parameters which are popped from the native argument stack. Pushes the return value which must be of type ptr.

N_CALL_R(N) (— ⇒ value)

Calls the function whose address is TOS. Uses the native calling convention for this platform with 0 parameters which are popped from the native argument stack. Pushes the return value which must be of type R.

N_CALL_U4(N) (— ⇒ value)

Calls the function whose address is TOS. Uses the native calling convention for this platform with 0 parameters which are popped from the native argument stack. Pushes the return value which must be of type uint32.

N_CALL_U8(N) (— ⇒ value)

Calls the function whose address is TOS. Uses the native calling convention for this platform with 0 parameters which are popped from the native argument stack. Pushes the return value which must be of type uint64.

N_CALL_V(N) (— ⇒ value)

Calls the function whose address is TOS. Uses the native calling convention for this platform with 0 parameters which are popped from the native argument stack. Pushes the return value which must be of type void.

OPCODE (— ⇒ opcode)

Pushes the current opcode to TOS.

OR_I4 (op1, op2 ⇒ value)

Binary operation: 32 bit signed integer bitwise or.

OR_I8 (op1, op2 ⇒ value)

Binary operation: 64 bit signed integer bitwise or.

OR_U4 (op1, op2 ⇒ value)

Binary operation: 32 bit unsigned integer bitwise or.

OR_U8 (op1, op2 ⇒ value)

Binary operation: 64 bit unsigned integer bitwise or.

PICK (— ⇒ $\mathbf{N^{th}}$)

1 operand byte.
Picks the N^{th} item from the stack(TOS is index 0)and pushes it onto the evaluation stack.

PLOAD_F4 (addr ⇒ value)

Load from memory. Push 32 bit floating point value loaded from address in TOS.

PLOAD_F8 (addr ⇒ value)

Load from memory. Push 64 bit floating point value loaded from address in TOS.

PLOAD_I1 (addr ⇒ value)

Load from memory. Push 8 bit signed integer value loaded from address in TOS.

PLOAD_I2 (addr ⇒ value)

Load from memory. Push 16 bit signed integer value loaded from address in TOS.

PLOAD_I4 (addr ⇒ value)

Load from memory. Push 32 bit signed integer value loaded from address in TOS.

PLOAD_I8 (addr ⇒ value)

Load from memory. Push 64 bit signed integer value loaded from address in TOS.

PLOAD_P (addr ⇒ value)

Load from memory. Push pointer value loaded from address in TOS.

PLOAD_R (addr ⇒ value)

Load from memory. Push reference value loaded from address in TOS.

PLOAD_U1 (addr ⇒ value)

Load from memory. Push 8 bit unsigned integer value loaded from address in TOS.

PLOAD_U2 (addr ⇒ value)

Load from memory. Push 16 bit unsigned integer value loaded from address in TOS.

PLOAD_U4 (addr ⇒ value)

Load from memory. Push 32 bit unsigned integer value loaded from address in TOS.

PLOAD_U8 (addr ⇒ value)

Load from memory. Push 64 bit unsigned integer value loaded from address in TOS.

PROTECT (— ⇒ value)

Pushes a new protect-object to the protection stack and pushes 0 to TOS, when initially executed. When execution resumes after a RAISE, the value that was TOS when the RAISE instruction was exceuted is pushed to TOS.

PROTECT_PUSH (— ⇒ protect)

Pops the protect-object from the protection stack.

PSTORE_F4 (value, array ⇒ —)

Store to memory. Store 32 bit floating point value in NOS to address in TOS.

PSTORE_F8 (value, array ⇒ —)

Store to memory. Store 64 bit floating point value in NOS to address in TOS.

PSTORE_I1 (value, array ⇒ —)

Store to memory. Store 8 bit signed integer value in NOS to address in TOS.

PSTORE_I2 (value, array ⇒ —)

Store to memory. Store 16 bit signed integer value in NOS to address in TOS.

PSTORE_I4 (value, array ⇒ —)

Store to memory. Store 32 bit signed integer value in NOS to address in TOS.

PSTORE_I8 (value, array ⇒ —)

Store to memory. Store 64 bit signed integer value in NOS to address in TOS.

PSTORE_P (value, array ⇒ —)

Store to memory. Store pointer value in NOS to address in TOS.

PSTORE_R (value, array ⇒ —)

Store to memory. Store reference value in NOS to address in TOS.

PSTORE_U1 (value, array ⇒ —)

Store to memory. Store 8 bit unsigned integer value in NOS to address in TOS.

PSTORE_U2 (value, array ⇒ —)

Store to memory. Store 16 bit unsigned integer value in NOS to address in TOS.

PSTORE_U4 (value, array ⇒ —)

Store to memory. Store 32 bit unsigned integer value in NOS to address in TOS.

PSTORE_U8 (value, array ⇒ —)

Store to memory. Store 64 bit unsigned integer value in NOS to address in TOS.

RAISE (value ⇒ —)

Pops the value from the stack. Unwinds the stack, if necessary, and resumes execution at the PROTECT instruction associated with the protect-object on top of the protection stack, pushing the value back to the restored stack.

RETURN_F4 (value ⇒ —)

Returns the 32 bit floating point value (on TOS).

RETURN_F8 (value ⇒ —)

Returns the 64 bit floating point value (on TOS).

RETURN_I4 (value ⇒ —)

Returns the 32 bit signed integer value (on TOS).

RETURN_I8 (value ⇒ —)

Returns the 64 bit signed integer value (on TOS).

RETURN_P (value ⇒ —)

Returns the pointer value (on TOS).

RETURN_R (value ⇒ —)

Returns the reference value (on TOS).

RETURN_U4 (value ⇒ —)

Returns the 32 bit unsigned integer value (on TOS).

RETURN_U8 (value ⇒ —)

Returns the 64 bit unsigned integer value (on TOS).

RETURN_V (value ⇒ —)

Returns

RLOAD_F4 (object, offset ⇒ value)

Load from object. Load 32 bit floating point value from object NOS at offset TOS.

RLOAD_F8 (object, offset ⇒ value)

Load from object. Load 64 bit floating point value from object NOS at offset TOS.

RLOAD_I1 (object, offset ⇒ value)

Load from object. Load 8 bit signed integer value from object NOS at offset TOS.

RLOAD_I2 (object, offset ⇒ value)

Load from object. Load 16 bit signed integer value from object NOS at offset TOS.

RLOAD_I4 (object, offset ⇒ value)

Load from object. Load 32 bit signed integer value from object NOS at offset TOS.

RLOAD_I8 (object, offset ⇒ value)

Load from object. Load 64 bit signed integer value from object NOS at offset TOS.

RLOAD_P (object, offset ⇒ value)

Load from object. Load pointer value from object NOS at offset TOS.

RLOAD_R (object, offset ⇒ value)

Load from object. Load reference value from object NOS at offset TOS.Any read-barriers required by the garbage collector are performed.

RLOAD_U1 (object, offset ⇒ value)

Load from object. Load 8 bit unsigned integer value from object NOS at offset TOS.

RLOAD_U2 (object, offset ⇒ value)

Load from object. Load 16 bit unsigned integer value from object NOS at offset TOS.

RLOAD_U4 (object, offset ⇒ value)

Load from object. Load 32 bit unsigned integer value from object NOS at offset TOS.

RLOAD_U8 (object, offset ⇒ value)

Load from object. Load 64 bit unsigned integer value from object NOS at offset TOS.

RSH_I4 (op1, op2 ⇒ value)

Binary operation: 32 bit signed integer right shift.

RSH_I8 (op1, op2 ⇒ value)

Binary operation: 64 bit signed integer right shift.

RSH_U4 (op1, op2 ⇒ value)

Binary operation: 32 bit unsigned integer right shift.

RSH_U8 (op1, op2 ⇒ value)

Binary operation: 64 bit unsigned integer right shift.

RSTORE_F4 (value, object, offset ⇒ —)

Store into object.Store 32 bit floating point value at 3OS into object NOS, offset TOS.

RSTORE_F8 (value, object, offset ⇒ —)

Store into object.Store 64 bit floating point value at 3OS into object NOS, offset TOS.

RSTORE_I1 (value, object, offset ⇒ —)

Store into object.Store 8 bit signed integer value at 3OS into object NOS, offset TOS.

RSTORE_I2 (value, object, offset ⇒ —)

Store into object.Store 16 bit signed integer value at 3OS into object NOS, offset TOS.

RSTORE_I4 (value, object, offset ⇒ —)

Store into object.Store 32 bit signed integer value at 3OS into object NOS, offset TOS.

RSTORE_I8 (value, object, offset ⇒ —)

Store into object.Store 64 bit signed integer value at 3OS into object NOS, offset TOS.

RSTORE_P (value, object, offset ⇒ —)

Store into object.Store pointer value at 3OS into object NOS, offset TOS.

RSTORE_R (value, object, offset ⇒ —)

Store into object.Store reference value at 3OS into object NOS, offset TOS.Any write-barriers required by the garbage collector are performed.

RSTORE_U1 (value, object, offset ⇒ —)

Store into object.Store 8 bit unsigned integer value at 3OS into object NOS, offset TOS.

RSTORE_U2 (value, object, offset ⇒ —)

Store into object.Store 16 bit unsigned integer value at 3OS into object NOS, offset TOS.

RSTORE_U4 (value, object, offset ⇒ —)

Store into object.Store 32 bit unsigned integer value at 3OS into object NOS, offset TOS.

RSTORE_U8 (value, object, offset ⇒ —)

Store into object.Store 64 bit unsigned integer value at 3OS into object NOS, offset TOS.

SIGN (val ⇒ extended)

Sign extend a single word to 64 bit. Leaves a double word in TOS and NOS for 32bit machines. This is a no-op for 64bit machines.

STACK (— ⇒ sp)

Pushes the Evaluation-stack stack-pointer to TOS. The evaluation stack grows downwards, so stack items will be at non-negative offsets from sp. Values pushed on to the stack are not visible, do *not* access values at negative offsets. As soon as a net positive number of values are popped from the stack, sp becomes invalid and should *not* be used.

SUB_F4 (op1, op2 ⇒ value)

Binary operation: 32 bit floating point subtract.

SUB_F8 (op1, op2 ⇒ value)

Binary operation: 64 bit floating point subtract.

SUB_I4 (op1, op2 ⇒ value)

Binary operation: 32 bit signed integer subtract.

SUB_I8 (op1, op2 ⇒ value)

Binary operation: 64 bit signed integer subtract.

SUB_P (op1, op2 ⇒ value)

Binary operation: pointer subtract.

SUB_U4 (op1, op2 ⇒ value)

Binary operation: 32 bit unsigned integer subtract.

SUB_U8 (op1, op2 ⇒ value)

Binary operation: 64 bit unsigned integer subtract.

SYMBOL (— ⇒ address)

2 operand bytes.
Push address of symbol to TOS

TARGET(N) (— ⇒ —)

Target for Jump and Branch.

TLOAD_F4(N) (— ⇒ value)

Push the contents of the Nth temporary variable as a 32 bit floating point.

TLOAD_F8(N) (— ⇒ value)

Push the contents of the Nth temporary variable as a 64 bit floating point.

TLOAD_I4(N) (— ⇒ value)

Push the contents of the Nth temporary variable as a 32 bit signed integer.

TLOAD_I8(N) (— ⇒ value)

Push the contents of the Nth temporary variable as a 64 bit signed integer.

TLOAD_P(N) (— ⇒ value)

Push the contents of the Nth temporary variable as a pointer.

TLOAD_R(N) (— ⇒ value)

Push the contents of the Nth temporary variable as a reference.

TLOAD_U4(N) (— ⇒ value)

Push the contents of the Nth temporary variable as a 32 bit unsigned integer.

TLOAD_U8(N) (— ⇒ value)

Push the contents of the Nth temporary variable as a 64 bit unsigned integer.

TSTORE_F4(N) (value ⇒ —)

Pop a 32 bit floating point from the stack and store in the Nth temporary variable.

TSTORE_F8(N) (value ⇒ —)

Pop a 64 bit floating point from the stack and store in the Nth temporary variable.

TSTORE_I4(N) (value ⇒ —)

Pop a 32 bit signed integer from the stack and store in the Nth temporary variable.

TSTORE_I8(N) (value ⇒ —)

Pop a 64 bit signed integer from the stack and store in the Nth temporary variable.

TSTORE_P(N) (value ⇒ —)

Pop a pointer from the stack and store in the Nth temporary variable.

TSTORE_R(N) (value ⇒ —)

Pop a reference from the stack and store in the Nth temporary variable.

TSTORE_U4(N) (value ⇒ —)

Pop a 32 bit unsigned integer from the stack and store in the Nth temporary variable.

TSTORE_U8(N) (value ⇒ —)

Pop a 64 bit unsigned integer from the stack and store in the Nth temporary variable.

TYPE_NAME(N,X) (— ⇒ —)

Name the (reference) type of the Nth temporary variable, for debugging purposes.

UNLOCK (lock ⇒ —)

Unlocks the fast-lock popped from TOS

UNLOCK_INTERNAL (offset, object ⇒ —)

Unlocks the fast-lock in object at TOS at offset NOS. Pops both object and offset from stack.

UNPROTECT (— ⇒ value)

Pops and destroys the protect-object on top of the exception stack.

V_CALL_F4 (— ⇒ value)

1 operand byte.
Variadic call. Calls the function whose address is TOS. Ensures that N parameters are removed from the evaluation stack and pushes the return value which must be of type float32. N is fetched from the instruction stream.

V_CALL_F8 (— ⇒ value)

1 operand byte.
Variadic call. Calls the function whose address is TOS. Ensures that N parameters are removed from the evaluation stack and pushes the return value which must be of type float64. N is fetched from the instruction stream.

V_CALL_I4 (— ⇒ value)

1 operand byte.
Variadic call. Calls the function whose address

is TOS. Ensures that N parameters are removed from the evaluation stack and pushes the return value which must be of type int32. N is fetched from the instruction stream.

V_CALL_I8 (— ⇒ value)

1 operand byte.
Variadic call. Calls the function whose address is TOS. Ensures that N parameters are removed from the evaluation stack and pushes the return value which must be of type int64. N is fetched from the instruction stream.

V_CALL_P (— ⇒ value)

1 operand byte.
Variadic call. Calls the function whose address is TOS. Ensures that N parameters are removed from the evaluation stack and pushes the return value which must be of type ptr. N is fetched from the instruction stream.

V_CALL_R (— ⇒ value)

1 operand byte.
Variadic call. Calls the function whose address is TOS. Ensures that N parameters are removed from the evaluation stack and pushes the return value which must be of type R. N is fetched from the instruction stream.

V_CALL_U4 (— ⇒ value)

1 operand byte.
Variadic call. Calls the function whose address is TOS. Ensures that N parameters are removed from the evaluation stack and pushes the return value which must be of type uint32. N is fetched from the instruction stream.

V_CALL_U8 (— ⇒ value)

1 operand byte.
Variadic call. Calls the function whose address is TOS. Ensures that N parameters are removed from the evaluation stack and pushes the return value which must be of type uint64. N is fetched from the instruction stream.

V_CALL_V (— ⇒ value)

1 operand byte.
Variadic call. Calls the function whose address is TOS. Ensures that N parameters are removed from the evaluation stack and pushes the return value which must be of type void. N is fetched from the instruction stream.

XOR_I4 (op1, op2 ⇒ value)

Binary operation: 32 bit signed integer bitwise exclusive or.

XOR_I8 (op1, op2 ⇒ value)

Binary operation: 64 bit signed integer bitwise exclusive or.

XOR_U4 (op1, op2 ⇒ value)

Binary operation: 32 bit unsigned integer bitwise exclusive or.

XOR_U8 (op1, op2 ⇒ value)

Binary operation: 64 bit unsigned integer bitwise exclusive or.

ZERO (val ⇒ extended)

Zero extend a single word to 64 bit. Leaves a double word in TOS and NOS for 32bit machines. This is a no-op for 64bit machines.