Managed Compiler Infrastructure

Release 1.0

The Lycus Foundation

June 06, 2013

CONTENTS

The MCI (Managed Compiler Infrastructure) is a modern and intuitive back end for compilers, runtimes, code analyzers, and other developer tools.

This is the infrastructure guide. It provides a high-level view of the features, architecture, and design of the MCI. It is intended to give the reader an overview of how the back end and virtual machine work. It is also a good reference for writing programs to run under the MCI's execution engines (the JIT/AOT compilers and the interpreter) and for the various command line tools that the MCI provides.

CHAPTER

ONE

INTRODUCTION

The MCI is a high-level, modern, and intuitive compiler back end written in the D 2.0 programming language. It has an intermediate representation that can easily model the concepts found in most managed languages today.

CHAPTER

TWO

TERMINOLOGY

This document attempts to explain various terms and abbreviations often used in the MCI source code and documentation.

2.1 AA

Abbreviation for alias analysis. This is the technique of proving whether or not two pointers definitely, definitely not, or possibly point to the same memory location.

2.2 ALU

Abbreviation for arithmetic logic unit. This refers to the unit in a processor which performs basic arithmetic and bit-wise operations. It usually includes operations such as addition, subtraction, multiplication, and division, but also several bit-wise operations such as the bit-wise AND, OR, XOR, etc.

2.3 AOT

An abbreviation for ahead of time. It generally refers to either the technique of compiling code before program execution, or such a compiler itself.

2.4 AST

An abstract tree-based representation of source code. Most parsers emit an AST from every parsed document, as this is usually the easiest kind of data structure to work with.

2.5 BB

Abbreviation for basic block.

2.6 Basic block

A basic block (or just block) is a set of instructions which, in SSA form, contains a number of simple instructions terminated by a single terminator instruction. If one were to compare with the C programming language, a basic block can be considered a label which a `goto` statement can transfer control to.

2.7 CSE

Abbreviation for common sub-expression elimination. This is an optimization which eliminates duplicate computations in expressions. For instance, in `x * y + x * y`, the computation of `x + y` can be factored out to a variable `z` such that the expression can be rewritten as `z + z`, thereby avoiding doing the computation of `x + y` twice.

2.8 DCE

Abbreviation for dead code elimination. This is an optimization that attempts to remove code that is definitely unreachable or otherwise considered useless (i.e. has no impact on the program's semantics). For instance:

```
x = 0;
// ...
if (x != 0)
{
    foo();
}
else
{
    bar();
}
```

It is trivial to discover that the true branch will never be taken. So, we optimize to:

```
x = 0;
// ...
bar();
```

Further optimization would remove `x` entirely.

2.9 EP

Abbreviation for entry point. An entry point of a main module is called upon startup and returns the exit code of the program.

2.10 FFI

Abbreviation for foreign function interface. It can either refer to the concept of calling a native function dynamically at runtime, or the actual action of doing so.

2.11 GC

An abbreviation for garbage collection (or garbage collector), which refers to the technique of using reachability analysis to determine whether memory should be freed, instead of placing this burden upon the programmer.

2.12 GC root

A GC root is a pointer which does not lie within the heap, and is used by the GC to start its reachability analysis from. This usually includes (but is not necessarily limited to) global fields, local registers, the program stack etc.

2.13 Heap

Refers to the data structure the operating system uses to manage its memory. In general, there are two heaps: The native heap and the managed heap. The former is what is usually accessed through LibC's `malloc()` and `free()` functions; the latter is the heap controlled by the GC.

2.14 IAL

Abbreviation for Intermediate Assembly Language. This is the IR used in the core of the MCI and is a four-address, linear representation.

It is usually in a static single assignment (SSA) form while in the analysis and optimization pipeline, but can also be in non-SSA form (for example, when doing native code generation or when executing in the interpreter).

2.15 Insn

Abbreviation for instruction.

2.16 Instr

Abbreviation for instruction.

2.17 IPA

Inter-procedural analysis. This is the practice of doing things like alias analysis and function inline cost analysis across function boundaries.

2.18 IPO

Inter-procedural optimization. This refers to optimizing across function boundaries, such as when inlining functions or doing global DCE.

2.19 IR

Abbreviation for intermediate representation. Computer programs are usually lowered to IRs to allow easier analysis and optimization for some specific tasks, but most importantly, in order to make native code generation easier.

Most IRs are in some kind of linear form, as it is hard to generate native code directly from a tree-based IR; linear code maps better to modern processors.

2.20 ISA

An abbreviation for instruction set architecture. This generally refers to the set of machine code instructions available in a processor architecture (and sometimes other features). It may also be used to describe the instruction set of IRs.

2.21 JIT

An abbreviation for just in time. It generally refers to either the technique of compiling code on demand, or such a compiler itself.

2.22 LTO

Link-time optimization. This is the practice of doing IPO across modules. As far as the MCI is concerned, this optimization comes for free, as all code must be available in IR form.

2.23 LibC

This is the standard library for the C programming language. It is typically exploited by many other languages, however, as it provides the easiest access to memory, I/O, and other such facilities which are very close to the operating system.

2.24 MCI

Abbreviation for Managed Compiler Infrastructure.

2.25 MEP

Abbreviation for module entry point. A module entry point is called once before any of the module's code is executed.

2.26 MXP

Abbreviation for module exit point. A module's exit point is called once when the program exits.

2.27 Main module

The main module of a program is the module that was passed to the virtual machine for execution.

2.28 PRE

Abbreviation for partial redundancy elimination. This is a form of CSE that tries to eliminate computations that are said to be partially redundant. For instance, consider this high-level code:

```
if (foo)
{
    x = y - 8;
}
else
{
    // ...
}
w = y - 8;
```

If `foo` is true, `y - 8` is evaluated twice. This is clearly wasteful, so this code can be optimized to:

```
z = y - 8;
if (foo)
{
    x = z;
}
else
{
    // ...
}
w = z;
```

2.29 RTO

An abbreviation for RuntimeObject. Refers to the runtime format and layout of values in the MCI, which generally consists of a type pointer, GC bits, and the user data field.

2.30 RTV

An abbreviation for RuntimeValue. Refers to a rooted object that holds a reference to a managed object.

2.31 SCCP

Abbreviation for sparse conditional constant propagation. An optimization performed in SSA form. It is strictly more powerful than applying DCE and constant propagation in any order or number of repetitions.

2.32 SSA

Abbreviation for static single assignment. This is a form of IR where variables are only assigned once, and so-called phi functions are used to determine which variable should be used depending on where control flow came from.

SSA is mostly useful in analysis and optimization.

2.33 TEP

Abbreviation for thread entry point. A thread entry point of a module is called before a properly registered thread executes any code within it.

2.34 TXP

Abbreviation for thread exit point. A thread exit point of a module is called whenever a properly registered thread is exited.

2.35 TLS

Abbreviation for thread-local storage. This is a mechanism by which each thread in a program gets its own isolated version of a variable.

2.36 Target

Refers to a processor architecture that the MCI can compile code for (therefore, a *target* for code generation). Examples include x86, ARM, etc.

2.37 Terminator

A terminator is an instruction which, while code is in SSA form, indicates the end of a basic block. Only one terminator is allowed in a basic block, and it must appear as the last instruction.

CHAPTER

THREE

COMMAND LINE TOOLS

The MCI provides a single command line application to access all command line tools. On most normal installations, this tool is simply called `mci`.

By running `mci -h`, you'll get an overview of the tools and parameters that are supported by the command line interface. This is only a short overview, though, and doesn't explain exactly how each tool is to be used. This page will shed some light on that.

3.1 General syntax

As running just `mci` suggests, the command line interface itself has only two options: `-h|--help` and `-s|--silent`. Generally, the short forms of the options are preferred, so those will be used in the rest of this document. The `-h` option simply displays the help overview and exits. The `-s` option makes the command line interface silent, i.e. it won't output anything to `stdout` and `stderr`. This is mostly useful if you're running a program in an execution engine and don't want the 'noise' that the command line interface normally outputs.

To run a tool, you simply pass its name and arguments to `mci`. So, for example, `mci asm foo.ial -o bar.mci`. This runs the assembler which parses `foo.ial` and generates a binary `bar.mci` which can be executed. In order to suppress output, you could also say `mci -s asm foo.ial -o bar.mci`.

3.2 Exit codes

The following primary exit codes can occur:

Value	Description
0	No errors.
1	A tool-specific error occurred.
2	Some part of command line processing failed.

Any other exit code is possible too; in particular, the execution engine tools will return the exit code of the hosted application. The above exit codes are just the ones that the command line interface is guaranteed to be able to return.

3.3 Tools

This section details the usage of the specific tools the command line interface supports. Note that all tool parameters are optional unless stated otherwise.

3.3.1 AOT compiler

Tool name `aot`

3.3.2 IAL assembler

Tool name `asm`

This tool assemblies IAL source files into an executable module. Other than the parameters it takes, all arguments are assumed to be IAL source file names. All IAL source files must end with the extension `.ial`.

The `-o` parameter specifies which file to write the resulting module to. The output file name must end in `.mci`. This defaults to `out.mci`.

The `-d` parameter specifies a dump file for the parsed ASTs. This is mostly useful for debugging, and not really for general usage.

3.3.3 Soft debugger

Tool name `dbg`

This tool runs an interactive soft debugger client on the command line. It allows you to connect to an execution engine with a running debugger server and interact with the program being executed.

3.3.4 IAL disassembler

Tool name `disasm`

Disassembles an assembled module to an IAL source file. It accepts one module file name only (must end in `.mci`). In general, this can be used to round-trip arbitrary IAL code.

The `-o` parameter specifies the output file. It must end in `.ial`. This defaults to `out.ial`.

3.3.5 Graph generator

Tool name `graph`

Generates a Graphviz control flow graph for a function. Takes as input exactly one module file name (must end in `.mci`) and one function name. This tool is mostly interesting if you are debugging internals of the MCI.

The `-o` parameter specifies the output file name. It must end in `.dot`. This defaults to `out.dot`.

3.3.6 Interpreter

Tool name `interp`

Executes a given module with the IAL interpreter. Accepts exactly one module file name (must end in `.mci`). The module must have a valid entry point function.

The `-c` parameter specifies which garbage collector should be used. See `mci -h` for possible values.

If the hosted program is started correctly, returns whatever exit code that program specified.

3.3.7 JIT compiler

Tool name `jit`

3.3.8 IAL linker

Tool name `link`

Links a set of modules into one module. Accepts a set of module file names as input (must end in `.mci`). If there are function or type name clashes, the selected resolution strategy is used to resolve them.

The `-r` parameter specifies which resolution strategy to use. See `mci -h` for possible values.

3.3.9 Linter

Tool name `lint`

Performs various static analyses for correctness on a set of modules. Accepts as input a set of module file names (must end in `.mci`).

These analyses are generally not very smart and can easily give false positives. They are primarily meant to help spot common errors in emitted IAL code. Note also that this tool only analyzes SSA functions.

3.3.10 Optimizer

Tool name `opt`

Optimizes a set of modules in place. Accepts as input a set of module file names (must end in `.mci`).

The `-p` option specifies an optimization pass to run. See `mci -h` for possible passes.

The `-1` parameter applies all fast optimization passes.

The `-2` parameter applies all moderate optimization passes.

The `-3` parameter applies all slow optimization passes.

Fast, moderate, and slow refer to the time it takes to run the passes.

Note that none of the parameters above imply any others, so passing e.g. `-2` does not imply `-1`.

The `-4` parameter applies all unsafe optimization passes. This allows some unsafe optimizations to happen which might change the actual semantics of the program. You should most likely not be using this.

Passes are applied in the exact order they are given on the command line (duplicate passes are OK and will be run repeatedly in the given order).

3.3.11 IAL verifier

Tool name `verify`

Verifies a set of modules for ISA and type system validity. Accepts as input a set of module file names (must end in `.mci`).

Note that a module must pass these verification passes in order for it to be executable in an execution engine.

3.3.12 Statistics

Tool name `stats`

Outputs statistics about a set of modules to `stdout`. Takes as input the file names of those modules (must end in `.mci`).

The `-g` parameter causes a list of global fields to be printed.

The `-e` parameter causes a list of thread fields to be printed.

The `-f` parameter causes a list of functions to be printed.

The `-t` parameter causes a list of types to be printed.

The `-d` parameter causes a list of data blocks to be printed.

CHAPTER

FOUR

OPTIMIZATION PASSES

This page lists the optimization passes that the MCI supports.

Fast passes are those that are considered extremely fast at executing, while slow passes are those that take a very long time to run. Moderate passes are somewhere in between.

Unsafe passes are those that may actually alter the semantics of a program in order to get better performance. In general, these should not be used unless you *really* know what you're doing (a program typically has to be written with these passes in mind in order to not break when they're used).

Note that some optimization passes only work in SSA form, while others only work on non-SSA form. Some passes are form-agnostic.

4.1 Fast passes

4.1.1 Nop remover

Pass name `nop-rem`

IR type Any

This pass removes all `nop` instructions in a function. It is mostly useful to conserve disk and memory space if a program has many `nop` instructions.

4.1.2 Comment remover

Pass name `comm-rem`

IR type Any

This pass removes all `comment` instructions in a function. It is mostly useful to conserve disk and memory space if a program has many `comment` instructions.

4.1.3 Unused register remover

Pass name `unused-reg`

IR type Any

This is a very simple pass that simply removes all unused registers in a function. This is completely harmless for the most part, but has the minor side-effect that the stack layout of the function will be different once unused registers are removed. Generally, programs should not rely on stack layout in the first place, so it is safe to assume that this optimization is always safe.

Running this pass after sparse conditional constant propagation and dead code elimination is generally a good idea, since it cleans up the registers left behind by those passes.

4.1.4 Unused basic block remover

Pass name `unused-bb`

IR type Any

This pass removes all unused basic blocks in a function. A basic block is considered unused if no branching instruction in the function targets it and the basic block isn't set as the unwind block of any *other* basic blocks (if the basic block has itself set as unwind block, it is considered unused).

Running this pass after sparse conditional constant propagation is generally a good idea, since it cleans up the basic blocks left behind by that pass, which can significantly reduce code size.

4.1.5 Constant folder

Pass name `const-fold`

IR type SSA

This pass performs simple constant folding. This includes all binary operators (like add, subtract, multiply, divide, and so on) except comparison operators. In general, the pass only concerns itself with integers with a fixed size and floating-point values. It doesn't attempt to optimize operations on native integers. Note also that the pass stops folding if it encounters a division by zero, since this usually means that a hardware trap must be generated at runtime, rather than silently ignoring it at compile time.

This pass should in most cases be applied before any other passes.

4.1.6 Dead code eliminator

Pass name `dce`

IR type SSA

This is an agressive dead code elimination pass. It assumes that all of a function's instructions are dead until proven otherwise.

Specifically, it starts out with a list of all 'root' instructions. These are the instructions known to be live unconditionally. The pass currently assumes, conservatively, that all instructions without a target register are live. Further, instructions with target registers that have side-effects (such as pinning a reference, allocating memory, and so on) are considered live. All terminator instructions are considered live as well. This list of root instructions is then used to propagate liveness backwards such that all of the instructions that the root instructions depend on are also considered live. Finally, the instructions that are not live are removed.

It's a good idea to run this after sparse conditional constant propagation to clean up dead definitions.

4.2 Moderate passes

4.3 Slow passes

4.4 Unsafe passes

CHAPTER

FIVE

ASSEMBLY LANGUAGE

Programs for the MCI can be written in the built-in assembly language, IAL (Intermediate Assembly Language). The assembler takes as input a series of source files and assembles them to a single output file (a module).

The grammar is:

```
Program  ::=   { TypeDeclaration | GlobalFieldDeclaration | ThreadFieldDeclaration | Fu
```

Module references have the grammar:

```
Module  ::=   Identifier
```

Some common grammar elements that will be used:

```
DecimalDigit                ::=   "0" ..  "9"
DecimalSequence             ::=   DecimalDigit { DecimalDigit }
HexadecimalDigit            ::=   DecimalDigit | "a" ..  "f" | "A" ..  "F"
HexadecimalSequence         ::=   HexadecimalDigit { HexadecimalDigit }
IdentifierCharacter         ::=   "."  | "_" | 'a' ..  'z' | 'A' ..  'Z'
Identifier                  ::=   IdentifierCharacter { IdentifierCharacter | DecimalDig
QuotedIdentifierCharacter   ::=   ?  any character ?  - "'" | "\'"
QuotedIdentifier            ::=   "'" QuotedIdentifierCharacter { QuotedIdentifierCharac
Literal                     ::=   [ "+" | "-" ] ( IntegerLiteral | FloatingPointLiteral
LiteralArray                ::=   Literal { "," Literal }
IntegerLiteral              ::=   DecimalSequence | "0x" HexadecimalSequence
FloatingPointLiteral        ::=   DecimalSequence "."  DecimalSequence [ "e" [ "+" | "-"
```

Line comments are allowed anywhere. They start with `//` and go until the end of the line, e.g.:

```
// This is a comment.
x = ari.add y, z; // Another comment.
```

5.1 Types

Structure types are aggregates of members. They can be used to form objects of strongly typed data, and can be allocated on the stack, the native heap, and on the GC-managed heap.

Type declarations have the grammar:

```
TypeDeclaration     ::=   [ MetadataList ] "type" Identifier [ AlignSpecification ] "{"
AlignSpecification  ::=   "align" Literal
```

The alignment specification can be used to override the automatic alignment algorithm that the MCI uses.

Type references have the grammar:

```
Type  ::=   [ Module "/" ] Identifier
```

The module reference is optional. If it is not specified, the type is looked up in the module being assembled.

The grammar for type specifications is:

```
ReturnType           ::=   "void" | TypeSpecification
TypeSpecification    ::=   CoreType | Type | PointerType | ReferenceType | ArrayType |
PointerType          ::=   TypeSpecification "*"
ReferenceType        ::=   TypeSpecification "&"
ArrayType            ::=   TypeSpecification "[" "]"
VectorType           ::=   TypeSpecification "[" Literal "]"
StaticArrayType      ::=   TypeSpecification "{" Literal "}"
FunctionPointerType  ::=   ReturnType "(" TypeParameterList ")" [ CallingConvention ]
TypeParameterList    ::=   "(" [ TypeSpecification { "," TypeSpecification } ] ")"
CoreType             ::=   "int" | "uint" | "int8" | "uint8" | "int16" | "uint16" | "in
```

5.1.1 Members

A member consists of a type and a name. Members are variables that represent the physical contents of structure types.

Member declarations have the grammar:

```
MemberDeclaration  ::=   [ MetadataList ] "field" TypeSpecification Identifier ";"
```

Member references have the grammar:

```
Member  ::=   Type ":"  Identifier
```

5.2 Fields

Fields that go into global or thread-local storage have the grammar:

```
GlobalFieldDeclaration     ::=   "field" "global" TypeSpecification Identifier ForeignF
ThreadFieldDeclaration     ::=   "field" "thread" TypeSpecification Identifier ForeignF
ForeignFieldSpecification  ::=   [ "(" ForeignSymbol ")" ]
```

Global fields are like global variables in C: They are shared across all threads in a process. Thread-local variables, on the other hand, get a unique instance per thread.

If a foreign field specification is given, the field is effectively a forward declaration for a field in another MCI module. It will be resolved and bound the first time it is used at runtime.

Field references have the grammar:

```
GlobalField  ::=   [ Module "/" ] Identifier
ThreadField  ::=   [ Module "/" ] Identifier
```

5.3 Functions

Functions are the MCI's answer to the procedure abstraction. A function takes a number of parameters as input and returns a single output value.

Function declarations have the grammar:

```
FunctionDeclaration  ::=   [ MetadataList ] "function" FunctionAttributes ReturnType Id
FunctionAttributes   ::=   [ "ssa" ] [ "pure" ] [ "nooptimize" ] [ "noinline" ] [ "nore
CallingConvention    ::=   "cdecl" | "stdcall"
FunctionBody         ::=   { RegisterDeclaration | BasicBlockDeclaration }
```

The `ssa` attribute specifies that the function is in SSA form. When a function is in SSA form, registers may only be assigned exactly once (i.e. using a register without assigning it is illegal), and must have an incoming definition before being used. The `copy` instruction is not allowed in SSA form. If a function is not in SSA form, the `phi` instruction is not allowed.

The `pure` attribute indicates that calls to the function can safely be reordered as the optimizer and code generator see fit. In other words, the function is referentially transparent: Calling it with the same arguments at any point in time will always yield the same result. This attribute should be used carefully, as incorrect use can result in wrong code generation.

The `nooptimize` flag indicates that a function must not be optimized. It will be ignored entirely by the optimization pipeline.

The `noinline` flag prevents a function from being inlined at call sites.

The `noreturn` flag indicates that a function does not return normally (e.g. by using `return` or `leave`). The optimization and code generation pipeline will assume that any code following a call to a `noreturn` function is effectively dead. Functions marked with `noreturn` are still allowed to throw exceptions, unless also marked `nothrow`.

The `nothrow` flag indicates that a function does not throw any exceptions. This property is transitive in the sense that all functions called by a `nothrow` function are also assumed to be `nothrow`. If a `nothrow` function does throw, behavior is undefined.

Function references have the grammar:

```
Function  ::=   [ Module "/" ] Identifier
```

The module reference is optional. If it is not specified, the function is looked up in the module being assembled.

5.3.1 Parameters

Parameters have the grammar:

```
ParameterList  ::=   "(" [ [ MetadataList ] Parameter { "," [ MetadataList ] Parameter
```

The `noescape` attribute only has significance for pointers, references, arrays, vectors, and function pointers. It indicates that the function will not escape an alias (i.e. pointer) to the pointed-to object. This means that the parameter is guaranteed to only reside in the current stack frame, or within objects that satisfy this same constraint.

5.3.2 Registers

A register consists of a type and a name. A function can have an arbitrary amount of registers. If a function is in SSA form, a register can only be assigned once, and is required to be assigned explicitly before use.

Registers are guaranteed to be completely zeroed out upon function entry.

Register declarations have the grammar:

```
RegisterDeclaration  ::=   [ MetadataList ] "register" TypeSpecification Identifier ";"
```

The grammar for a register reference is:

```
Register  ::=   Identifier
```

5.3.3 Basic blocks

A basic block is a linear sequence of instructions, containing exactly one terminator instruction at the end. This terminator instruction can branch to other basic blocks, return from the function, etc.

Basic block declarations have the grammar:

```
BasicBlockDeclaration  ::=   [ MetadataList ] "block" ( "entry" | Identifier ) [ Unwind
UnwindSpecification    ::=   "unwind" BasicBlock
```

The unwind specification is a basic block reference and specifies where to unwind to if an exception is thrown within the basic block.

The grammar for a basic block reference is:

```
BasicBlock  ::=   "entry" | Identifier
```

Instructions

Instructions encode the actual logic of a program. They're contained linearly in basic blocks.

Their grammar is:

```
Instruction            ::=   [ MetadataList ] InstructionAttributes [ Register "=" ] ?
InstructionAttributes  ::=   [ "volatile" ]
InstructionOperand     ::=   "(" ( Literal | LiteralArray | BasicBlock | BranchTarget |
BranchTarget           ::=   BasicBlock "," BasicBlock
RegisterSelector       ::=   Register { "," Register }
ForeignSymbol          ::=   Identifier "," Identifier
```

The full list of valid instructions (with register counts, operand types, and so on) can be found on the instruction set page. Note that the parser is driven by that information; for example, if an instruction requires a field reference as operand, the parser will expect to be able to parse one.

The `volatile` attribute ensures that an instruction is not reordered (by the optimization pipeline and code generator) relative to other volatile instructions. Further, instructions that seem dead (a store followed by a store to the exact same location, for example) will not be optimized out. This is useful to model the semantics of the `volatile` qualifier in the C family of languages. Note that it has nothing to do with concurrency.

Some attributes only have meaning for certain instructions. For example, the `volatile` attribute has no meaning for instructions that don't involve memory accesses. Meaningless attributes are allowed on instructions but optimizers are free to remove them. The linter will also warn about them.

5.4 Data blocks

Data blocks are blobs of arbitrary data:

```
DataBlockDeclaration  ::=  "data" Identifier "(" LiteralArray ")" ";"
```

Data blocks consist of a series of unsigned 8-bit bytes. They can contain any data at all. They hold no particular meaning as far as the MCI is concerned.

Data block references have the grammar:

```
DataBlock  ::=  [ Module "/" ] Identifier
```

5.5 Entry points

An entry point can be specified for a module. If this is done, the module effectively becomes executable as a program.

The grammar is:

```
EntryPointDeclaration  ::=  "entry" Function ";"
```

An entry point function must return `int32`, have no parameters, and have standard calling convention.

A module entry point can be specified. It will be called before any code inside the module is executed at all and/or any loads, stores, and address-of operations on static/TLS fields in the module.

The grammar is:

```
ModuleEntryPointDeclaration  ::=  "module" "entry" Function ";"
```

A module exit point can also be specified. It will be called once a program has returned from its main entry point.

The grammar is:

```
ModuleExitPointDeclaration  ::=  "module" "exit" Function ";"
```

Module entry and exit points must have no parameters, return `void`, and have standard calling convention.

Module entry and exit points will only be called once during a program's execution time. A module's module exit point is only guaranteed to be called if that module's module entry point was ever called during execution time.

Module entry points are guaranteed to be called before any thread entry points. Module exit points are guaranteed to be called after any thread exit points.

A thread entry point can also be specified. Such an entry point is guaranteed to run before a properly registered thread gets a chance to execute any other managed code inside the module. This is useful for initializing TLS data.

The grammar is:

```
ThreadEntryPointDeclaration  ::=  "thread" "entry" Function ";"
```

A thread entry point function must return `void`, have no parameters, and have standard calling convention.

Note that thread entry points may be invoked concurrently if multiple threads enter the virtual machine at the same time. The same holds true for thread exit points when threads exit.

Thread exit points are also available to help tear down TLS data. They are guaranteed to be called just before a thread exits, and will only be called once the thread has stopped executing any other managed code.

The grammar is:

```
ThreadExitPointDeclaration  ::=   "thread" "exit" Function ";"
```

As with thread entry points, these must return `void`, have no parameters, and have standard calling convention.

A module's thread exit point is only guaranteed to be called if that module's thread entry point has been called.

A module can only have one entry point, one thread entry point, one thread exit point, one module entry point, and one module exit point (all are optional). They must refer to functions inside the module.

Normally, thread entry and exit points and module entry and exit points will only be called whenever some thread attempts to access code (or fields) inside the module they belong to. Some execution engines may, however, choose to load all of a program's modules eagerly, resulting in these entry and exit points being executed even if no code inside their module was executed during the program's execution time.

Code inside thread entry and exit points and module entry and exit points must not make any assumptions about the order they are called in. The order will for all practical purposes be deterministic, but this is by no means guaranteed.

5.6 Metadata

Metadata can be attached to type declarations, field declarations, function declarations, register declarations, basic block declarations, and instructions.

The grammar is:

```
MetadataList  ::=   "[" MetadataPair { "," MetadataPair } "]"
MetadataPair  ::=   Identifier ":"  Identifier
```

Metadata is mostly useful to the optimizer and compiler pipeline.

CHAPTER

SIX

TYPE SYSTEM

The MCI usus a mostly strong, nominal type system. The type system consists of the following categories of types:

- Primitive types: Integer and floating-point types (`int32`, `int64`, `float32`, `float64`, etc).
- Structure types: Similar to `structs` in C.
- Type specifications: These are said to have one or more "element types".
  - Pointer types: Ye olde `int32*` and so on.
  - Reference types: Similar to pointers, but can only refer to structure types, and may only have one indirection (for example, `Foo&`).
  - Array types: Simple one-dimensional arrays with a dynamic length (for example, `float64[]`).
  - Vector types: Similar to arrays, but they have a fixed, static length (i.e. `float64[3]`).
  - Static array types: Similar to vectors, but live 'in place' where they are used (i.e. in a structure or a register). For example, `int{3}`.
  - Function pointer types: These point to a function which can be invoked indirectly. They contain a calling convention, return type and parameter types (for example, `int32(float32, float64)` would be a pointer to a function taking a `float32` and a `float64` argument, returning `int32`).

The following notation is used:

Notation	Meaning
`T`	Type name.
`T[]`	Array of `T`.
`T[E]`	Vector of `T` with `E` elements.
`T{E}`	Static array of `T` with `E` elements.
`T*`	Pointer to `T`.
`T&`	Reference to `T`.
`R(T1, ...)`	Function pointer returning `R`, taking `T1`, ... arguments.
`R(T1, ...) cdecl`	Function pointer with `cdecl` calling convention.

6.1 Primitive types

These are the building blocks of any application; they are the most basic data types and represent integers and floating-point values. The following primitive types exist:

- `int8`: 8-bit signed integer.
- `uint8`: 8-bit unsigned integer.

- `int16`: 16-bit signed integer.
- `uint16`: 16-bit unsigned integer.
- `int32`: 32-bit signed integer.
- `uint32`: 32-bit unsigned integer.
- `int64`: 64-bit signed integer.
- `uint64`: 64-bit unsigned integer.
- `int`: Native-size signed integer (32-bit or 64-bit).
- `uint`: Native-size unsigned integer (32-bit or 64-bit).
- `float32`: 32-bit IEEE 754 floating-point value.
- `float64`: 64-bit IEEE 754 floating-point value.

The fixed-width integers and floating-point types are guaranteed to be the same size on all platforms. `int` and `uint` will be 32 or 64 bits wide depending on the pointer length of the platform.

All primitives are convertible to/from each other.

6.2 Structure types

A structure is a record that encapsulates a fixed number of fields, each of their own type. A field consists of a type and a name.

Examples:

```
// A structure with two instance fields. These can be accessed on any
// instance of Foo, both as a value instance or as a pointer with one
// indirection.
type Foo
{
    field int32 bar;
    field float64 baz;
}
```

A structure can also specify its alignment (this is normally decided by the compiler). The alignment must either be zero or a power of two. If it is zero, the compiler picks the alignment (that is to say, zero is like the default). Examples:

```
// Use automatic alignment.
type Foo3 align 0
{
}

// Align fields sequentially.
type Foo4 align 1
{
}

// Align fields on a boundary of 16 bytes.
type Foo5 align 16
{
}
```

Structures can be created in several ways:

- On the stack as a value: Simply declare a register typed as the structure. This makes it live on the stack with value semantics, and it will not participate in any kind of dynamic memory allocation.
- On the stack, dynamically allocated: Declare a register as a pointer to the structure and allocate the memory with `mem.salloc` or `mem.snew`.
- On the heap, dynamically allocated: Declare a register as either a pointer to the structure, or as a vector or array of the structure. Then, allocate memory with `mem.alloc` or `mem.new`.
- On the heap, GC-tracked: Declare a register as a reference to the structure and allocate an instance with `mem.new`. Additionally, references can be contained in vectors and arrays, and in other GC-tracked structures.

6.3 Type specifications

Type specifications are types that contain or encapsulate other types, such as pointers, arrays, vectors, etc.

6.3.1 Pointer types

A pointer is, semantically, just a native-size integer pointing to some location in memory where the real value is. A pointer can point to any other type (including pointers, resulting in several indirections).

Examples:

- Pointer to `int32`: `int32*`
- Pointer to array of `float32`: `float32[]*`
- Pointer to pointer to `uint`: `uint**`

Pointers are convertible to any other pointer type (including function pointers) and the primitives `int` and `uint`.

6.3.2 Reference types

References are similar to pointers, but are tracked by the GC (vectors and arrays are also references, but this is implicit).

It is important to note that a reference value must be aligned on a native word-size boundary. For example, this is problematic:

```
type BadAlign align 1
{
    field uint8 a;

    // This field will now be unaligned. This is undefined behavior.
    field BadAlign& b;
}
```

Care should be taken when using an explicit alignment specification on structures that contain references. The MCI's garbage collector, optimizer, and code generator all assume that reference fields are aligned.

In addition to this rule, the object that the reference points to must be on a native word-size boundary as well. This is less important to users, as the `mem.new` instruction guarantees this.

Structures instantiated on the GC heap are prefixed by a header (which is implementation-defined) containing type information, GC bits, and so on. This header also has a dedicated native word-sized field that can be accessed with `field.user.addr`. This field is primarily there to let compilers assign language-specific type information to objects.

Examples of references:

- Reference to a structure called Foo: `Foo&`

Any reference-to-reference conversion is valid, including reference-to-array and reference-to-vector conversions.

6.3.3 Array types

These are single-dimensional, length-aware collections of elements. The exact start and end of an array in memory is undefined, but all elements are guaranteed to be laid out contiguously. In other words, an array can be iterated by fetching the address of the first element and incrementing the pointer.

The elements of an array are guaranteed to start at a boundary suitable for SIMD operations on the machine. This typically means on an 8-byte, 16-byte, or 32-byte boundary, depending on the architecture (and the target machine's detected features). The exact alignment should, for all practical purposes, be considered undefined, however.

Reading beyond the bounds of an array results in undefined behavior.

Arrays can only be allocated as GC-tracked objects.

Examples:

- Array of `int32`: `int32[]`
- Array of pointers to `float64`: `float64*[]`
- Array of arrays of `int8`: `int8[][]`

Any array-to-array/vector conversion is valid as long as the source array's element type is convertible to the target array/vector's element type.

6.3.4 Vector types

Vectors are similar to arrays in that they contain a series of contiguous elements. Vectors, however, have a fixed, static length. This makes them very easy to use with vectorization technology such as SIMD, as the JIT compiler can unroll the SIMD operations statically.

Reading beyond the bounds of a vector results in undefined behavior.

Vectors can only be allocated as GC-tracked objects.

Examples:

- Vector of `int32` with 3 elements: `int32[3]`
- Vector of pointers to `int32` with 64 elements: `int32*[64]`
- Vector of 3 vectors of `int32` with 8 elements: `int32[8][3]`

Any vector-to-vector/array conversion is valid as long as the source vector's element type is convertible to the target vector/array's element type.

6.3.5 Static array types

Static arrays are similar to vectors with the difference that they are stored 'in place'. That is, if a field in a structure is typed to be a static array, that array's elements will be embedded directly in the structure. A register typed to be a static array will also result in the the entire array being on the stack.

Static arrays are, like arrays and vectors, guaranteed to be suitably aligned for SIMD operations on the machine.

Static arrays are passed by value. This is unlike the C calling convention where they are passed by reference. The same behavior can be achieved by simply passing pointers to static arrays.

Examples:

- Static array of `int32` with 3 elements: `int32{3}`
- Static array of pointers to `int32` with 64 elements: `int32*{64}`
- Static array of 3 static arrays of `int32` with 8 elements: `int32{8}{3}`

Static arrays cannot be converted to any other type.

6.3.6 Function pointer types

These are simply pointers to functions in memory. A function pointer carries information about the calling convention, return type, and parameter types. Calling convention is optional; if it is not specified, the default IAL calling convention is assumed.

Equality between function pointers pointing to the same function is guaranteed if the function pointers are loaded using `load.func`. All other guarantees are up to the operating system the code is running on.

Examples:

- Function returning `int32`, taking no parameters: `int32()`
- Function returning `void` (i.e. nothing), taking `float32`: `void(float32)`
- Function returning `void`, taking `float32` and `int32`: `void(float32, int32)`
- Function returning `void`, taking no parameters, with `cdecl` calling convention: `void() cdecl`

Function pointers are convertible to any pointer type (including other function pointer types).

CHAPTER

SEVEN

INSTRUCTION SET

This page describes the instruction set used in the IAL ISA.

7.1 Utility instructions

These instructions serve no particular purpose as far as execution goes, but are useful for annotating the instruction stream.

7.1.1 nop

Has target register No

Source registers 0

Operand type None

Performs no actual operation. This can be useful to mark regions of code that will be patched later in the compilation process.

7.1.2 comment

Has target register No

Source registers 0

Operand type 8-bit unsigned integer array

Similar to nop, but allows attaching arbitrary data to it. Note that when the MCI displays the data, it assumes it to be encoded as UTF-8 text.

7.2 Constant load instructions

These instructions load constant values into registers.

7.2.1 load.i8

Has target register Yes

Source registers 0

Operand type 8-bit signed integer

Loads a constant 8-bit signed integer into the target register.

The target register must be of type `int8`.

7.2.2 load.ui8

Has target register Yes

Source registers 0

Operand type 8-bit unsigned integer

Loads a constant 8-bit unsigned integer into the target register.

The target register must be of type `uint8`.

7.2.3 load.i16

Has target register Yes

Source registers 0

Operand type 16-bit signed integer

Loads a constant 16-bit signed integer into the target register.

The target register must be of type `int16`.

7.2.4 load.ui16

Has target register Yes

Source registers 0

Operand type 16-bit unsigned integer

Loads a constant 16-bit unsigned integer into the target register.

The target register must be of type `uint16`.

7.2.5 load.i32

Has target register Yes

Source registers 0

Operand type 32-bit signed integer

Loads a constant 32-bit signed integer into the target register.

The target register must be of type `int32`.

7.2.6 load.ui32

Has target register Yes

Source registers 0

Operand type 32-bit unsigned integer

Loads a constant 32-bit unsigned integer into the target register.

The target register must be of type `uint32`.

7.2.7 load.i64

Has target register Yes

Source registers 0

Operand type 64-bit signed integer

Loads a constant 64-bit signed integer into the target register.

The target register must be of type `int64`.

7.2.8 load.ui64

Has target register Yes

Source registers 0

Operand type 64-bit unsigned integer

Loads a constant 64-bit unsigned integer into the target register.

The target register must be of type `uint64`.

7.2.9 load.f32

Has target register Yes

Source registers 0

Operand type 32-bit floating-point value

Loads a constant 32-bit floating-point value into the target register.

The target register must be of type `float32`.

7.2.10 load.f64

Has target register Yes

Source registers 0

Operand type 64-bit floating-point value

Loads a constant 64-bit floating-point value into the target register.

The target register must be of type `float64`.

7.2.11 load.i8a

Has target register Yes

Source registers 0

Operand type 8-bit signed integer array

Loads a constant array of 8-bit signed integers into the target register.

The target register must be of type `int8[]`, `int8*`, or a vector or static array of `int8` with an element count matching that of the array operand.

When the target register is a pointer, the data must be explicitly freed with mem.free. If the given array is of zero length, a null pointer is assigned to the target register.

7.2.12 load.ui8a

Has target register Yes

Source registers 0

Operand type 8-bit unsigned integer array

Loads a constant array of 8-bit unsigned integers into the target register.

The target register must be of type `uint8[]`, `uint8*`, or a vector or static array of `uint8` with an element count matching that of the array operand.

When the target register is a pointer, the data must be explicitly freed with mem.free. If the given array is of zero length, a null pointer is assigned to the target register.

7.2.13 load.i16a

Has target register Yes

Source registers 0

Operand type 16-bit signed integer array

Loads a constant array of 16-bit signed integers into the target register.

The target register must be of type `int16[]`, `int16*`, or a vector or static array of `int16` with an element count matching that of the array operand.

When the target register is a pointer, the data must be explicitly freed with mem.free. If the given array is of zero length, a null pointer is assigned to the target register.

7.2.14 load.ui16a

Has target register Yes

Source registers 0

Operand type 16-bit unsigned integer array

Loads a constant array of 16-bit unsigned integers into the target register.

The target register must be of type `uint16[]`, `uint16*`, or a vector or static array of `uint16` with an element count matching that of the array operand.

When the target register is a pointer, the data must be explicitly freed with mem.free. If the given array is of zero length, a null pointer is assigned to the target register.

7.2.15 load.i32a

Has target register Yes

Source registers 0

Operand type 32-bit signed integer array

Loads a constant array of 32-bit signed integers into the target register.

The target register must be of type `int32[]`, `int32*`, or a vector or static array of `int32` with an element count matching that of the array operand.

When the target register is a pointer, the data must be explicitly freed with mem.free. If the given array is of zero length, a null pointer is assigned to the target register.

7.2.16 load.ui32a

Has target register Yes

Source registers 0

Operand type 32-bit unsigned integer array

Loads a constant array of 32-bit unsigned integers into the target register.

The target register must be of type `uint32[]`, `uint32*`, or a vector or static array of `uint32` with an element count matching that of the array operand.

When the target register is a pointer, the data must be explicitly freed with mem.free. If the given array is of zero length, a null pointer is assigned to the target register.

7.2.17 load.i64a

Has target register Yes

Source registers 0

Operand type 64-bit signed integer array

Loads a constant array of 64-bit signed integers into the target register.

The target register must be of type `int64[]`, `int64*`, or a vector or static array of `int64` with an element count matching that of the array operand.

When the target register is a pointer, the data must be explicitly freed with mem.free. If the given array is of zero length, a null pointer is assigned to the target register.

7.2.18 load.ui64a

Has target register Yes

Source registers 0

Operand type 64-bit unsigned integer array

Loads a constant array of 64-bit unsigned integers into the target register.

The target register must be of type `uint64[]`, `uint64*`, or a vector or static array of `uint64` with an element count matching that of the array operand.

When the target register is a pointer, the data must be explicitly freed with mem.free. If the given array is of zero length, a null pointer is assigned to the target register.

7.2.19 load.f32a

Has target register Yes

Source registers 0

Operand type 32-bit floating-point value array

Loads a constant array of 32-bit floating-point values into the target register.

The target register must be of type `float32[]`, `float32*`, or a vector or static array of `float32` with an element count matching that of the array operand.

When the target register is a pointer, the data must be explicitly freed with mem.free. If the given array is of zero length, a null pointer is assigned to the target register.

7.2.20 load.f64a

Has target register Yes

Source registers 0

Operand type 64-bit floating-point value array

Loads a constant array of 64-bit floating-point values into the target register.

The target register must be of type `float64[]`, `float64*`, or a vector or static array of `float64` with an element count matching that of the array operand.

When the target register is a pointer, the data must be explicitly freed with mem.free. If the given array is of zero length, a null pointer is assigned to the target register.

7.2.21 load.func

Has target register Yes

Source registers 0

Operand type Function reference

Loads a function pointer to the given function into the target register.

The target register must be of a function pointer type with a signature that matches the function reference. For example, a function declared as:

```
function int32 foo(float32, float64)
{
    ...
}
```

can be assigned to a register declared as:

```
register int32(float32, float64) bar;
```

The target may also have a specified calling convention (`cdecl` or `stdcall`), in which case the given function must have a matching calling convention.

Equality for function pointers obtained through this instruction is guaranteed. That is, if a function pointer to a specific function is loaded twice, the two pointers are guaranteed to be equal. Ordering is, however, not guaranteed.

7.2.22 load.null

Has target register Yes

Source registers 0

Operand type None

Loads a null value into the target register.

The target register must be a pointer, a function pointer, an array, a vector, or a reference.

7.2.23 load.size

Has target register Yes

Source registers 0

Operand type Type specification

Loads the absolute size of a type specification's layout in memory into the target register.

Note that for vectors, this is not the full size of the vector, but rather the size of the reference to the vector (as with arrays and pointers). For static arrays, this is the full size of the entire array.

The target register must be of type `uint`.

7.2.24 load.align

Has target register Yes

Source registers 0

Operand type Type specification

Loads the alignment of a type specification into the target register.

The target register must be of type `uint`.

7.2.25 load.offset

Has target register Yes

Source registers 0

Operand type Member reference

Loads the offset of a field in its containing structure type into the target register.

The target register must be of type `uint`.

load.data

Has target register Yes

Source registers 0

Operand type Data block reference

Loads a pointer to a data block into the target register. The pointer should never be explicitly freed and is always valid.

The target register must be of type `uint8*`.

7.3 Arithmetic and logic instructions

These instructions provide the basic ALU.

7.3.1 ari.add

Has target register Yes

Source registers 2

Operand type None

Adds the value in the first source register to the value in the second source register and stores the result in the target register.

This instruction can have one of two forms:

- All three registers must be of the exact same type. Allowed types are `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `int64`, `uint64`, `int`, `uint`, `float32`, and `float64`. This performs regular arithmetic.
- The target register is a pointer type. The first source register must also be a pointer type, and the second source register must be `uint`. This performs pointer arithmetic.

7.3.2 ari.sub

Has target register Yes

Source registers 2

Operand type None

Subtracts the value in the first source register from the value in the second source register and stores the result in the target register.

This instruction can have one of two forms:

- All three registers must be of the exact same type. Allowed types are `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `int64`, `uint64`, `int`, `uint`, `float32`, and `float64`. This performs regular arithmetic.
- The target register is a pointer type. The first source register must also be a pointer type, and the second source register must be `uint`. This performs pointer arithmetic.

7.3.3 ari.mul

Has target register Yes

Source registers 2

Operand type None

Multiplies the value in the first source register with the value in the second source register and stores the result in the target register.

All three registers must be of the exact same type. Allowed types are `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `int64`, `uint64`, `int`, `uint`, `float32`, and `float64`.

7.3.4 ari.div

Has target register Yes

Source registers 2

Operand type None

Divides the value in the first source register by the value in the second source register and stores the result in the target register.

If the divisor is zero and the computation involves integers, behavior is undefined. For floating-point types, behavior depends on the machine.

All three registers must be of the exact same type. Allowed types are `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `int64`, `uint64`, `int`, `uint`, `float32`, and `float64`.

7.3.5 ari.rem

Has target register Yes

Source registers 2

Operand type None

Computes the remainder resulting from dividing the first source register with the second source register and stores the result in the target register.

If the divisor is zero and the computation involves integers, behavior is undefined. For floating-point types, behavior depends on the machine.

All three registers must be of the exact same type. Allowed types are `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `int64`, `uint64`, `int`, `uint`, `float32`, and `float64`.

7.3.6 ari.neg

Has target register Yes

Source registers 1

Operand type None

Negates the value in the source register and assigns the result to the target register.

Both registers must be of the exact same type. Allowed types are `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `int64`, `uint64`, `int`, `uint`, `float32`, and `float64`.

7.3.7 bit.and

Has target register Yes

Source registers 2

Operand type None

Performs a bit-wise AND operation on the two source registers and assigns the result to the target register.

All three registers must be of the exact same type. Allowed types are `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `int64`, `uint64`, `int`, and `uint`.

7.3.8 bit.or

Has target register Yes

Source registers 2

Operand type None

Performs a bit-wise OR operation on the two source registers and assigns the result to the target register.

All three registers must be of the exact same type. Allowed types are `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `int64`, `uint64`, `int`, and `uint`.

7.3.9 bit.xor

Has target register Yes

Source registers 2

Operand type None

Performs a bit-wise XOR operation on the two source registers and assigns the result to the target register.

All three registers must be of the exact same type. Allowed types are `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `int64`, `uint64`, `int`, and `uint`.

7.3.10 bit.neg

Has target register Yes

Source registers 1

Operand type None

Performs a bit-wise complement negation operation on the source register and assigns the result to the target register.

Both registers must be of the exact same type. Allowed types are `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `int64`, `uint64`, `int`, and `uint`.

7.3.11 not

Has target register Yes

Source registers 1

Operand type None

Performs a logical negation operation on the source register and assigns the result to the target register.

If the source equals 0, the result is 1. In all other cases, the result is 0.

Both registers must be of the exact same type. Allowed types are `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `int64`, `uint64`, `int`, `uint`, `float32`, and `float64`.

7.3.12 shl

Has target register Yes

Source registers 2

Operand type None

Shifts the bits of the first source register to the left by the amount given in the second source register and assigns the result to the target register.

If the second source register is larger than or equal to the amount of bits of the first source register's type, behavior is undefined.

The first register and the target register must be of the exact same type. Allowed types are `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `int64`, `uint64`, `int`, and `uint`.

The second register must be of type `uint`.

7.3.13 shr

Has target register Yes

Source registers 2

Operand type None

Shifts the bits of the first source register to the right by the amount given in the second source register and assigns the result to the target register.

If the type of the values being shifted is signed, the shift is an arithmetic shift (i.e. it is done with sign extension); otherwise, a logical shift is done (i.e. zero extension is used).

If the second source register is larger than or equal to the amount of bits of the first source register's type, behavior is undefined.

The first register and the target register must be of the exact same type. Allowed types are `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `int64`, `uint64`, `int`, and `uint`.

The second register must be of type `uint`.

7.3.14 rol

Has target register Yes

Source registers 2

Operand type None

Rotates the bits of the value in the first source register left by the amount given in the second source register. This is similar to shl, but instead of performing zero extension, the rotated bits are inserted.

If the second source register is larger than or equal to the amount of bits of the first source register's type, behavior is undefined.

The first register and the target register must be of the exact same type. Allowed types are `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `int64`, `uint64`, `int`, and `uint`.

The second register must be of type `uint`.

7.3.15 ror

Has target register Yes

Source registers 2

Operand type None

Rotates the bits of the value in the first source register right by the amount given in the second source register. This is similar to shr, but instead of performing zero/sign extension, the rotated bits are inserted.

If the second source register is larger than or equal to the amount of bits of the first source register's type, behavior is undefined.

The first register and the target register must be of the exact same type. Allowed types are `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `int64`, `uint64`, `int`, and `uint`.

The second register must be of type `uint`.

7.4 Memory management instructions

These instructions are used to allocate and free memory from the system. There are instructions that operate on the native heap and others that operate on the GC-managed heap.

7.4.1 mem.alloc

Has target register Yes

Source registers 1

Operand type None

Allocates memory from either the native heap (if the target register is a pointer) or from the GC currently in use (if the target register is an array).

The source register indicates how many elements to allocate memory for. This means that if the target register is a pointer, the total amount of memory allocated is the size of the target register's element type times the element count. Otherwise, it represents the amount of array elements to be allocated. The source register must be of type `uint`.

If the target register is a pointer and the source register holds a zero value, the target register is set to a null pointer. For the array case, a zero-sized array will be allocated.

If the requested amount of memory could not be allocated, a null pointer is assigned to the target register; otherwise, the pointer to the allocated memory is assigned.

If the allocation was successful, all allocated memory is guaranteed to be completely zeroed out.

The target register must be a pointer or an array.

7.4.2 mem.new

Has target register Yes

Source registers 0

Operand type None

Allocates memory from the native heap (if the target register is a pointer) or from the GC currently in use (if the target register is a reference or a vector).

This operation allocates memory for a single fixed-size value. Thus, the the amount of memory allocated is the size of the element type of the target register (for vectors, this includes all elements).

If the requested amount of memory could not be allocated, a null pointer is assigned to the target register; otherwise, the pointer to the allocated memory is assigned.

If the allocation was successful, all allocated memory is guaranteed to be completely zeroed out.

The target register must be a pointer, a reference, or a vector.

7.4.3 mem.free

Has target register No

Source registers 1

Operand type None

Frees the memory pointed to by a pointer previously allocated with either mem.alloc or mem.new.

If the pointer passed in is null, no operation is performed. If the pointer is in some way invalid (e.g. it points to the interior of a block of allocated memory or has never been allocated in the first place), undefined behavior occurs.

This instruction deallocates from the right heap depending on the type of the source register (i.e. the GC-managed heap for arrays, vectors, and references, and the native heap for pointers).

The source register must be a pointer, a reference, an array, or a vector.

When invoking this instruction on a reference, an array, or a vector, it is assumed that the object being freed is only live in the source register, and absolutely nowhere else in the program. This makes this instruction very dangerous to use for managed objects. It is undefined behavior to use memory that has been freed.

7.4.4 mem.salloc

Has target register Yes

Source registers 1

Operand type None

Similar to mem.alloc. This instruction, however, allocates the memory on the stack. This means that memory allocated with this instruction shall not be freed manually with mem.free, as the code generator inserts cleanup code automatically.

As with mem.alloc, this instruction assigns a null pointer if the source register holds a value of zero.

If a stack overflow occurs in the allocation, behavior is undefined.

The source register must be of type `uint`.

The target register must be a pointer.

7.4.5 mem.snew

Has target register Yes

Source registers 0

Operand type None

Similar to mem.new. This instruction, however, allocates the memory on the stack. This means that memory allocated with this instruction shall not be freed manually with mem.free, as the code generator inserts cleanup code automatically.

If a stack overflow occurs in the allocation, behavior is undefined.

The target register must be a pointer.

7.4.6 mem.pin

Has target register Yes

Source registers 1

Operand type None

Pins a reference previously allocated with mem.new or mem.alloc so that the object it points to cannot be relocated by a compacting GC. This is useful when calling into external code via ffi, as the GC cannot track GC-managed memory beyond managed code. This also implies that the memory which is pinned will never be collected until it is unpinned. Therefore, memory leaks can happen if care is not taken to correctly mem.unpin the memory.

Passing a null or already-pinned reference to this instruction results in undefined behavior. The resulting value of this instruction is an opaque handle which only has meaning to the specific GC implementation. The handle is intended for use with mem.unpin later.

The source register must be a reference, an array, or a vector.

The target register must be of type `uint`.

7.4.7 mem.unpin

Has target register No

Source registers 1

Operand type None

Unpins memory previously pinned with mem.pin. The source register must be a handle returned by mem.pin. Any invalid handle value will result in undefined behavior (this includes handles already unpinned).

Care should be taken to only unpin the memory once it is certain that the memory is no longer referenced outside managed code. Failure to ensure this can result in undefined behavior.

7.5 Memory aliasing instructions

These instructions can be used for general pointer manipulation, such as dereferencing, setting memory values, etc.

7.5.1 mem.get

Has target register Yes

Source registers 1

Operand type None

Dereferences the pointer in the source register and assigns the resulting element value to the target register.

If the dereference operation failed in some way (e.g. the source pointer is null or points to invalid memory), undefined behavior occurs.

Dereferencing function pointers is not possible. Doing so by casting a function pointer to a regular pointer results in undefined behavior.

The source register must be a pointer, while the target register must be the element type of the source register's pointer type.

7.5.2 mem.set

Has target register No

Source registers 2

Operand type None

Sets the value of the memory pointed to by the pointer in the first register to the value of the second register.

If the memory addressing operation failed in some way (e.g. the target pointer is null or points to invalid memory), undefined behavior occurs.

Setting the pointed-to value of function pointers is not possible. Doing so by casting a function pointer to a regular pointer results in undefined behavior.

The first register must be a pointer type, while the second register must be the element type of the first register's pointer type.

7.5.3 mem.addr

Has target register Yes

Source registers 1

Operand type None

Takes the address of the value in the source register and assigns the address to the target register.

Dereferencing or writing to the resulting address once the current stack frame is no longer valid will result in undefined behavior.

The source register can be of any type, while the target register must be a pointer to the source register's type.

7.6 Array and vector instructions

These instructions are used to index into and manipulate arrays and vectors.

7.6.1 array.addr

Has target register Yes

Source registers 2

Operand type None

Retrieves the address to the element given in the second source register of the array given in the first source register and assigns it to the target register.

If the source array/vector is null, behavior is undefined. Taking the address of an element beyond the bounds of an array is acceptable, but dereferencing or writing to it results in undefined behavior.

The first source register must be an array, vector, or static array, while the second register must be of type `uint`.

The target register must be a pointer to the first source register's element type.

7.6.2 array.len

Has target register Yes

Source registers 1

Operand type None

Loads the length of an array into the target register. For arrays, this is the dynamic size, while for vectors and static arrays, it is the fixed size.

If the source array/vector is null, behavior is undefined.

The source register must be an array, vector, or static array.

The target register must be of type `uint`.

7.6.3 array.ari.add

Has target register No

Source registers 3

Operand type None

Performs arithmetic addition on elements of arrays, vectors, or static arrays.

If any of the involved arrays/vectors are null, undefined behavior occurs.

The first two source registers must be arrays, vectors, or static arrays of the types allowed in ari.add, and must have the same element type.

If the first source register is an array or vector of a pointer type, the third source register must either be of type `uint` or an array or vector of these. Otherwise, the third source register must be of the element type of the first source register, or be an array or vector of the first source register's element type.

7.6.4 array.ari.sub

Has target register No

Source registers 3

Operand type None

Performs arithmetic subtraction on elements of arrays, vectors, or static arrays.

If any of the involved arrays/vectors are null, undefined behavior occurs.

The first two source registers must be arrays, vectors, or static arrays of the types allowed in ari.sub, and must have the same element type.

If the first source register is an array or vector of a pointer type, the third source register must either be of type `uint` or an array or vector of these. Otherwise, the third source register must be of the element type of the first source register, or be an array or vector of the first source register's element type.

7.6.5 array.ari.mul

Has target register No

Source registers 3

Operand type None

Performs arithmetic multiplication on elements of arrays, vectors, or static arrays.

If any of the involved arrays/vectors are null, undefined behavior occurs.

The first two source registers must be arrays, vectors, or static arrays of the types allowed in ari.mul, and must have the same element type.

The third source register must be of the element type of the first source register, or be an array or vector of the first source register's element type.

7.6.6 array.ari.div

Has target register No

Source registers 3

Operand type None

Performs arithmetic division on elements of arrays, vectors, or static arrays.

If any of the involved arrays/vectors are null, undefined behavior occurs. If the divisor is zero and the computation involves integers, behavior is undefined. For floating-point types, behavior depends on the machine.

The first two source registers must be arrays, vectors, or static arrays of the types allowed in ari.div, and must have the same element type.

The third source register must be of the element type of the first source register, or be an array or vector of the first source register's element type.

7.6.7 array.ari.rem

Has target register No

Source registers 3

Operand type None

Computes the remainder resulting from dividing elements of arrays, vectors, or static arrays with the given value(s).

If any of the involved arrays/vectors are null, undefined behavior occurs. If the divisor is zero and the computation involves integers, behavior is undefined. For floating-point types, behavior depends on the machine.

The first two source registers must be arrays, vectors, or static arrays of the types allowed in ari.rem, and must have the same element type.

The third source register must be of the element type of the first source register, or be an array or vector of the first source register's element type.

7.6.8 array.ari.neg

Has target register No

Source registers 2

Operand type None

Negates all elements of an array, vector, or static array.

If any of the involved arrays/vectors are null, undefined behavior occurs.

The two source registers must be arrays, vectors, or static arrays of the types allowed in ari.neg, and must have the same element type.

7.6.9 array.bit.and

Has target register No

Source registers 3

Operand type None

Performs bit-wise AND on elements of arrays, vectors, or static arrays.

If any of the involved arrays/vectors are null, undefined behavior occurs.

The first two source registers must be arrays, vectors, or static arrays of the types allowed in bit.and, and must have the same element type.

The third source register must be of the element type of the first source register, or be an array or vector of the first source register's element type.

7.6.10 array.bit.or

Has target register No

Source registers 3

Operand type None

Performs bit-wise OR on elements of arrays, vectors, or static arrays.

If any of the involved arrays/vectors are null, undefined behavior occurs.

The first two source registers must be arrays, vectors, or static arrays of the types allowed in bit.or, and must have the same element type.

The third source register must be of the element type of the first source register, or be an array or vector of the first source register's element type.

7.6.11 array.bit.xor

Has target register No

Source registers 3

Operand type None

Performs bit-wise XOR on elements of arrays, vectors, or static arrays.

If any of the involved arrays/vectors are null, undefined behavior occurs.

The first two source registers must be arrays, vectors, or static arrays of the types allowed in bit.xor, and must have the same element type.

The third source register must be of the element type of the first source register, or be an array or vector of the first source register's element type.

7.6.12 array.bit.neg

Has target register No

Source registers 2

Operand type None

Performs a bit-wise complement negation operation on all elements of an array, vector, or static array.

If any of the involved arrays/vectors are null, undefined behavior occurs.

The two source registers must be arrays, vectors, or static arrays of the types allowed in bit.neg, and must have the same element type.

7.6.13 array.not

Has target register No

Source registers 2

Operand type None

Performs a logical negation on all elements of an array, vector, or static array.

If any of the involved arrays/vectors are null, undefined behavior occurs.

The first two source registers must be arrays, vectors, or static arrays of the types allowed in not, and must have the same element type.

7.6.14 array.shl

Has target register No

Source registers 3

Operand type None

Performs a left shift of the bits of elements in an array, vector, or static array.

If any of the involved arrays/vectors are null, undefined behavior occurs. If the shift amount is larger than or equal to the amount of bits of the element types involved, behavior is undefined.

The first two source registers must be arrays, vectors, or static arrays of the types allowed in shl, and must have the same element type.

The third source register must be of type `uint` or an array, vector, or static array of these.

7.6.15 array.shr

Has target register No

Source registers 3

Operand type None

Performs a right shift of the bits of elements in an array, vector, or static array.

If any of the involved arrays/vectors are null, undefined behavior occurs. If the shift amount is larger than or equal to the amount of bits of the element types involved, behavior is undefined.

The first two source registers must be arrays, vectors, or static arrays of the types allowed in shr, and must have the same element type.

The third source register must be of type `uint` or an array, vector, or static array of these.

7.6.16 array.rol

Has target register No

Source registers 3

Operand type None

Performs a left rotation of bits of the elements in an array, vector, or static array.

If any of the involved arrays/vectors are null, undefined behavior occurs. If the shift amount is larger than or equal to the amount of bits of the element types involved, behavior is undefined.

The first two source registers must be arrays, vectors, or static arrays of the types allowed in rol, and must have the same element type.

The third source register must be of type `uint` or an array, vector, or static array of these.

7.6.17 array.ror

Has target register No

Source registers 3

Operand type None

Performs a right rotation of bits of the elements in an array, vector, or static array.

If any of the involved arrays/vectors are null, undefined behavior occurs. If the shift amount is larger than or equal to the amount of bits of the element types involved, behavior is undefined.

The first two source registers must be arrays, vectors, or static arrays of the types allowed in ror, and must have the same element type.

The third source register must be of type `uint` or an array, vector, or static array of these.

7.6.18 array.conv

Has target register No

Source registers 2

Operand type None

Converts elements in the array, vector, or static array in the first source register to the element type of the array, vector, or static array in the second source register and assigns them to the second source register's elements incrementally.

The following conversions are valid:

- `T[]` -> `U[]` for any valid `T` -> `U` conversion.
- `T[]` -> `U[F]` for any valid `T` -> `U` conversion.
- `T[]` -> `U{F}` for any valid `T` -> `U` conversion.
- `T[E]` -> `U[]` for any valid `T` -> `U` conversion.
- `T[E]` -> `U[F]` for any valid `T` -> `U` conversion.
- `T[E]` -> `U{F}` for any valid `T` -> `U` conversion.
- `T{E}` -> `U[]` for any valid `T` -> `U` conversion.
- `T{E}` -> `U[F]` for any valid `T` -> `U` conversion.
- `T{E}` -> `U{F}` for any valid `T` -> `U` conversion.

If any of the involved arrays/vectors are null, undefined behavior occurs.

See also conv.

7.6.19 array.cmp.eq

Has target register No

Source registers 3

Operand type None

Performs a cmp.eq on all elements of arrays, vectors, or static arrays.

If any of the involved arrays/vectors are null, undefined behavior occurs.

The first source register must be an array, vector, or static array of `uint`. The second and third source registers must be arrays, vectors, or static arrays having the same element type.

7.6.20 array.cmp.neq

Has target register No

Source registers 3

Operand type None

Performs a cmp.neq on all elements of arrays, vectors, or static arrays.

If any of the involved arrays/vectors are null, undefined behavior occurs.

The first source register must be an array, vector, or static array of `uint`. The second and third source registers must be arrays, vectors, or static arrays having the same element type.

7.6.21 array.cmp.gt

Has target register No

Source registers 3

Operand type None

Performs a cmp.gt on all elements of arrays, vectors, or static arrays.

If any of the involved arrays/vectors are null, undefined behavior occurs.

The first source register must be an array, vector, or static array of `uint`. The second and third source registers must be arrays, vectors, or static arrays having the same element type.

7.6.22 array.cmp.lt

Has target register No

Source registers 3

Operand type None

Performs a cmp.lt on all elements of arrays, vectors, or static arrays.

If any of the involved arrays/vectors are null, undefined behavior occurs.

The first source register must be an array, vector, or static array of `uint`. The second and third source registers must be arrays, vectors, or static arrays having the same element type.

7.6.23 array.cmp.gteq

Has target register No

Source registers 3

Operand type None

Performs a cmp.gteq on all elements of arrays, vectors, or static arrays.

If any of the involved arrays/vectors are null, undefined behavior occurs.

The first source register must be an array, vector, or static array of `uint`. The second and third source registers must be arrays, vectors, or static arrays having the same element type.

7.6.24 array.cmp.lteq

Has target register No

Source registers 3

Operand type None

Performs a cmp.lteq on all elements of arrays, vectors, or static arrays.

If any of the involved arrays/vectors are null, undefined behavior occurs.

The first source register must be an array, vector, or static array of `uint`. The second and third source registers must be arrays, vectors, or static arrays having the same element type.

7.7 Structure field instructions

These instructions are used to operate on fields contained in structures types and pointers to them.

7.7.1 field.addr

Has target register Yes

Source registers 1

Operand type Member reference

Gets the address of the field given as the operand on the structure given in the source register and assigns it to the target register.

If the source register is a reference or a pointer, and is null, behavior is undefined.

Note that if the given structure is in a register with no indirection (i.e. on the stack), dereferencing and writing to the pointer's address when the current stack frame is no longer valid results in undefined behavior. Also, if the given structure is a reference, the resulting pointer is effectively an interior pointer. This means that reading and writing the memory it points to is only valid while the object it points into is live. Reading or writing to its address when the object is no longer live results in undefined behavior.

The source register must be a structure or a pointer or reference to a structure with at most one indirection.

The target register must be a pointer to the type of the field given in the operand.

7.7.2 field.user.addr

Has target register Yes

Source registers 1

Operand type None

Fetches the address of the source register's header user data field and assigns it to the target register.

If the source register is null, behavior is undefined.

Note that, since the resulting address is effectively an interior pointer, it will only be recognized by the GC in roots. Dereferencing the pointer or writing to its address is only legal while the object it points into is live. Reading or writing to its address when the object is no longer live results in undefined behavior.

The source register must be a reference, an array, or a vector.

The target register must be a pointer to either a reference, an array, or a vector.

7.7.3 field.global.addr

Has target register Yes

Source registers 0

Operand type Global field reference

Similar to field.addr, but operates on global fields. This means that the instruction does not need an instance of the structure to set the value of the given field.

Pointers to global fields are always valid.

7.7.4 field.thread.addr

Has target register Yes

Source registers 0

Operand type Global field reference

Similar to field.addr, but operates on TLS fields. This means that the instruction does not need an instance of the structure to set the value of the given field.

Pointers to TLS fields are valid so long as the thread owning the field instance that a pointer is pointing to has not exited.

7.8 Comparison instructions

These instructions test relativity of their source registers.

7.8.1 cmp.eq

Has target register Yes

Source registers 2

Operand type None

Compares the two source registers for equality. If they are equal, the target register is set to 1; otherwise, 0.

The source registers must be of the exact same type, and can be one of `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `int64`, `uint64`, `int`, `uint`, `float32`, `float64`, or any pointer type (in which case the pointers are compared for equality).

The target register must be of type `uint`.

7.8.2 cmp.neq

Has target register Yes

Source registers 2

Operand type None

Compares the two source registers for inequality. If they are unequal, the target register is set to 1; otherwise, 0.

The source registers must be of the exact same type, and can be one of `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `int64`, `uint64`, `int`, `uint`, `float32`, `float64`, or any pointer type (in which case the pointers are compared for equality).

The target register must be of type `uint`.

7.8.3 cmp.gt

Has target register Yes

Source registers 2

Operand type None

Determines if the value in the first source register is greater than the value in the second source register. If this is true, the target register is set to 1; otherwise, 0.

The source registers must be of the exact same type, and can be one of `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `int64`, `uint64`, `int`, `uint`, `float32`, `float64`, or any pointer type (in which case the pointers are compared).

The target register must be of type `uint`.

7.8.4 cmp.lt

Has target register Yes

Source registers 2

Operand type None

Determines if the value in the first source register is lesser than the value in the second source register. If this is true, the target register is set to 1; otherwise, 0.

The source registers must be of the exact same type, and can be one of `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `int64`, `uint64`, `int`, `uint`, `float32`, `float64`, or any pointer type (in which case the pointers are compared).

The target register must be of type `uint`.

7.8.5 cmp.gteq

Has target register Yes

Source registers 2

Operand type None

Determines if the value in the first source register is greater than or equal to the value in the second source register. If this is true, the target register is set to 1; otherwise, 0.

The source registers must be of the exact same type, and can be one of `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `int64`, `uint64`, `int`, `uint`, `float32`, `float64`, or any pointer type (in which case the pointers are compared).

The target register must be of type `uint`.

7.8.6 cmp.lteq

Has target register Yes

Source registers 2

Operand type None

Determines if the value in the first source register is lesser than or equal to the value in the second source register. If this is true, the target register is set to 1; otherwise, 0.

The source registers must be of the exact same type, and can be one of `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `int64`, `uint64`, `int`, `uint`, `float32`, `float64`, or any pointer type (in which case the pointers are compared).

The target register must be of type `uint`.

7.9 Function invocation instructions

These instructions are used to call functions and function pointers.

7.9.1 arg.push

Has target register No

Source registers 1

Operand type None

Enqueues the value in the source register into the functiona call argument queue.

This instruction must be immediately followed by another arg.push or any of call, call.tail, call.indirect, invoke, invoke.tail, or invoke.indirect.

The type of the value must equal the type of the function parameter at the same index as this instruction.

7.9.2 arg.pop

Has target register Yes

Source registers 0

Operand type None

Dequeues an argument given to a function. This instruction can only appear in the `entry` basic block of a function, and must either be the first instruction or come right after a previous arg.pop.

The target register must match the type of the function parameter at the same index as this instruction.

7.9.3 call

Has target register Yes

Source registers 0

Operand type Function reference

This performs a call to the function given as operand. This instruction expects that the function has a return type (i.e. it does not return `void`).

This instruction should follow immediately after a correct sequence of arg.push instructions.

The result (as returned by the called function) is assigned to the target register.

The target register's type must match the given function's return type.

7.9.4 call.tail

Has target register Yes

Source registers 0

Operand type Function reference

Works exactly like a call, except that this instruction hints to the code generator that tail call optimization must be done.

This instruction must be immediately followed by a return instruction which must return the resulting value of this call.

Tail calls can only be done in functions with standard calling convention.

7.9.5 call.indirect

Has target register Yes

Source registers 1

Operand type None

Performs a function call like the call instruction, but indirectly.

This instruction must (like call) be immediately preceeded by a correct arg.push sequence matching the function pointer's signature.

If the given function pointer is null or does not point to a valid function entry point, behavior is undefined.

The result of the call is assigned to the target register.

The source register must be a function pointer to a function returning non-`void`, and the target register must match the function pointer's return type.

7.9.6 invoke

Has target register No

Source registers 0

Operand type Function reference

This instruction does the same thing as call, but only works for functions with no return type (i.e. returning `void`), and thus has no target register.

7.9.7 invoke.tail

Has target register No

Source registers 0

Operand type Function reference

This instruction does the same thing as call.tail, but only works for functions with no return type (i.e. returning `void`), and thus has no target register.

This instruction must be immediately followed by a leave instruction.

Tail calls can only be done in functions with standard calling convention.

7.9.8 invoke.indirect

Has target register No

Source registers 1

Operand type None

This instruction does the same thing as call.indirect, but only works for function pointers with no return type (i.e. returning `void`), and thus has no target register.

7.10 Control flow instructions

These instructions are used to transfer control from one point in a program to another. Most are generally terminator instructions.

7.10.1 jump

Has target register No

Source registers 0

Operand type Basic block

Performs an unconditional jump to the specified basic block.

This is a terminator instruction.

7.10.2 jump.cond

Has target register No

Source registers 1

Operand type Branch selector

Performs a jump to the first basic block if the value in the source register does not equal 0; otherwise, jumps to the second basic block.

The source register must be of type `uint`.

This is a terminator instruction.

7.10.3 leave

Has target register No

Source registers 0

Operand type None

Leaves (i.e. returns from) the current function. This is only valid if the function returns `void` (or, in other words, has no return type).

Using this instruction in a `noreturn` function results in undefined behavior.

This is a terminator instruction.

7.10.4 return

Has target register No

Source registers 1

Operand type None

Returns from the current function with the value in the source register as the return value. This is only valid in functions that don't return `void` (i.e. have a return type).

Using this instruction in a `noreturn` function results in undefined behavior.

The source register must be the exact same type as the function's return type.

This is a terminator instruction.

7.10.5 dead

Has target register No

Source registers 0

Operand type None

Informs the optimizer of a branch that can safely be assumed unreachable (and thus optimized out). Any code following this instruction is assumed to be dead.

This is a terminator instruction.

7.10.6 phi

Has target register Yes

Source registers 0

Operand type Register selector

This instruction is used while the code is in SSA form. Due to the nature of SSA, it is often necessary to determine which register to use based on where control flow came from. This instruction picks the register which was assigned in the basic block control flow entered from and assigns it to the target register.

This instruction is valid only during analysis and optimization. It must not appear in code passed to the interpreter or JIT/AOT engines.

The target register and selector registers must all be of the same type.

Note that this instruction doesn't count as a control flow instruction. That is to say, multiple phi instructions are allowed in a basic block while in SSA form, and they do not act as terminators.

7.10.7 raw

Has target registers No

Source registers 0

Operand type 8-bit unsigned integer array

This instruction tells the code generator to insert raw machine code (which is given as the byte array operand) in the generated machine code stream. This must be the only instruction in a raw function.

This instruction has a few consequences:

- It must be the only instruction in the function.
- The function must have `cdecl` or `stdcall` calling convention.
- All optimizations that would affect the layout of the stack cannot happen.

Of course, usage of this instruction results in unportable code.

This instruction is primarily intended to allow the implementation of inline assembly in high-level languages. Arguments given to raw functions are passed according to the calling convention of the function and the return value (if any) should be passed according to the calling convention too.

It should be noted that this is not sufficient to implement full-blown inline assembly as in many C and C++ compilers. A general requirement of inline assembly using this instruction is that the raw blob must contain code that is neutral to relocations, as it is not in any way guaranteed where the code blob will be emitted in memory.

If the raw machine code returns and the function is marked `noreturn`, undefined behavior results.

This is a terminator instruction.

7.10.8 ffi

Has target register No

Source registers 0

Operand type Foreign function

This instruction marks the function as an FFI function. FFI functions must only contain this one instruction, which points the code generator to the actual function entry point in a native library.

This instruction has a few consequences:

- It must be the only instruction in the function.
- The function must have `cdecl` or `stdcall` calling convention.

Note that the native function isn't linked to statically. The execution engine (either the interpreter or the JIT/AOT engines) will attempt to locate the native entry point when the FFI function is called.

If the native function returns and the function is marked `noreturn`, undefined behavior results.

This is a terminator instruction.

7.10.9 forward

Has target register No

Source registers 0

Operand type Foreign function

This instruction marks a function as a reference. This means that, when a call to the function containing this instruction is made, the execution engine will forward it to a function in another module, as specified in the signature in the operand.

The operand must point to a function with standard calling convention in a managed MCI module. The module name should not contain the file extension; only the base name.

The function, when located in the specified module, is expected to have the exact same return type, parameter types, and attributes as the function this instruction is used in. If it does not, a runtime error results.

Note that using forwarded functions results in some classes of optimizations (e.g. inlining) being disabled for calls to such functions.

This is a terminator instruction.

7.11 Exception handling instructions

These are used to indicate and handle errors.

7.11.1 eh.throw

Has target register No

Source registers 1

Operand type None

Throws an exception. This causes the runtime to unwind the stack until an appropriate unwind block is found. If an unwind block is found, control transfers to that block. If none is found, the program is terminated.

If the given reference is null, behavior is undefined.

The source register must be a reference.

This is a terminator instruction.

7.11.2 eh.rethrow

Has target register No

Source registers 0

Operand type None

Rethrows an in-flight exception. This is different from using `eh.throw` to rethrow an exception reference in that this instruction does not reset the stack trace.

This instruction may only appear in unwind blocks.

This is a terminator instruction.

7.11.3 eh.catch

Has target register Yes

Source registers 0

Operand type None

This catches the current in-flight exception and assigns it to the target register. Note that this is not type-safe; it's similar to casting one reference type to another with `conv`. In order to determine the exact exception type, language/ABI-specific checks must be made.

This instruction may only appear in unwind blocks.

The target register must be a reference.

7.12 Miscellaneous instructions

Instructions that don't quite fit anywhere else.

7.12.1 copy

Has target register Yes

Source registers 1

Operand type None

This instruction copies the value in the source register into the target register. This is similar to a simple assignment in most programming languages; it is not a deep copy.

This instruction is not valid in SSA form.

The source register's type must match the target register's type.

7.12.2 conv

Has target register Yes

Source registers 1

Operand type None

Converts the value in the source register from one type to another, and assigns the resulting value to the target register.

The following conversions are valid:

- `T` -> `U` for any primitives `T` and `U` (`int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `int64`, `uint64`, `int`, `uint`, `float32`, and `float64`).
- `T*` -> `U*` for any `T` and any `U`.
- `T*` -> `uint` or `int` for any `T`.
- `uint` or `int` -> `T*` for any `T`.
- `T` -> `U` for any managed types (reference, array, or vector) `T` and `U`.
- `R1(T1, ...)` -> `R2(U1, ...)` for any `R1`, any `R2`, and any amount and type of T_n and U_m.
- `R(T1, ...)` -> `U*` for any `R`, any amount and type of T_n, and any `U`.
- `T*` -> `R(U1, ...)` for any `T`, any `R`, and any amount and type of U_n.

7.12.3 fence

Has target register No

Source registers 0

Operand type None

Inserts a full read/write memory barrier. This ensures that all loads and stores prior to this instruction will always be executed before loads and stores following this instruction. This is particularly useful in lock-free data structures and similar low-level constructs.

7.12.4 tramp

Has target register Yes

Source registers 1

Operand type None

Constructs a trampoline for a given function pointer. Trampolines are useful if the function pointer is to be passed to external code (e.g. via ffi) which might use the function pointer in threads not registered with the MCI. The generated trampoline will ensure that such an external thread is correctly registered before allowing it to call into managed code.

The source register must be any function pointer type. The target register must be a function pointer type with `cdecl` or `stdcall` calling convention matching the parameters and return type of the source register.

CHAPTER

EIGHT

INTRINSICS

The MCI defines a number of built-in functions that can be called by any program compiled with the infrastructure. These all reside in the `mci` module, which is actually implemented in D code inside the `mci.vm` library.

All intrinsics are thread safe.

This module is given special treatment by the assembler, so you do not need to provide a physical module that implements it.

8.1 Types

8.1.1 Object

This is an opaque type which is useful for representing an arbitrary reference type:

```
type Object
{
}
```

8.1.2 Weak

This is an opaque wrapper type given special treatment by garbage collector implementations that support it. It facilitates so-called weak references:

```
type Weak
{
}
```

Instances of this type should not be manipulated directly. The layout of this type is completely unspecified and any reliance on it is unsupported. To work with instances of `Weak`, use the related intrinsics.

8.2 Configuration information

These intrinsics retrieve information about the environment the MCI was compiled in.

8.2.1 get_compiler

Signature `uint8 get_compiler()`

Gets a value indicating which compiler was used to build the MCI.

Possible values:

Value	Description
0	Unknown compiler.
1	Digital Mars D (DMD).
2	GNU D Compiler (GDC).
3	LLVM D Compiler (LDC).

8.2.2 get_architecture

Signature `uint8 get_architecture()`

Gets a value indicating which architecture the MCI was compiled for.

Possible values:

Value	Description
0	x86 (32-bit or 64-bit).
1	ARM (32-bit).
2	PowerPC (32-bit or 64-bit).
3	Itanium (64-bit).
4	MIPS (32-bit or 64-bit).

8.2.3 get_operating_system

Signature `uint8 get_operating_system()`

Gets a value indicating which operating system the MCI was compiled on.

Possible values:

Value	Description
0	All Windows systems.
1	All Linux systems.
2	Mac OS X (and other Darwin systems).
3	FreeBSD.
4	Solaris.
5	AIX.

8.2.4 get_endianness

Signature `uint8 get_endianness()`

Gets a value indicating which endianness the MCI was compiled for.

Possible values:

Value	Description
0	Little endian.
1	Big endian.

8.2.5 get_emulation_layer

Signature `uint8 get_emulation_layer()`

Gets a value indicating which emulation layer the MCI is compiled under.

Possible values:

Value	Description
0	No emulation layer.
1	Cygwin.
2	MinGW.

8.2.6 is_32_bit

Signature `uint is_32_bit()`

Gets a value indicating whether the MCI is compiled for 32-bit pointers.

This function returns 0 if the MCI is compiled for 64-bit pointers; 1 if it's compiled for 32-bit pointers.

8.3 Atomic operations

8.3.1 atomic_load

Signature `Object& atomic_load(Object&*)`

Atomically loads the reference from the memory location pointed to by the first argument.

Full sequential consistency is guaranteed.

8.3.2 atomic_store

Signature `void atomic_store(Object&*, Object&)`

Atomically sets the location pointed to by the first argument to the reference in the second argument.

Full sequential consistency is guaranteed.

8.3.3 atomic_exchange

Signature `uint atomic_exchange(Object&*, Object&, Object&)`

Stores the reference in the third argument to the location pointed to by the first argument if the reference pointed to by the first argument is equal to the second argument. All of this happens atomically.

Returns 1 if the store happened; otherwise, returns 0.

Full sequential consistency is guaranteed.

8.3.4 atomic_load_u

Signature `uint atomic_load_u(uint*)`

Atomically loads the value from the memory location pointed to by the first argument.

Full sequential consistency is guaranteed.

8.3.5 atomic_store_u

Signature `void atomic_store_u(uint*, uint)`

Atomically sets the location pointed to by the first argument to the value in the second argument.

Full sequential consistency is guaranteed.

8.3.6 atomic_exchange_u

Signature `uint atomic_exchange_u(uint*, uint, uint)`

Stores the value in the third argument to the location pointed to by the first argument if the value pointed to by the first argument is equal to the second argument. All of this happens atomically.

Returns 1 if the store happened; otherwise, returns 0.

Full sequential consistency is guaranteed.

8.3.7 atomic_add_u

Signature `uint atomic_add_u(uint*, uint)`

Atomically adds the value in the second argument to the value pointed to by the first argument and returns the result.

The result is also assigned to the location pointed to by the first argument.

Full sequential consistency is guaranteed.

8.3.8 atomic_sub_u

Signature `uint atomic_sub_u(uint*, uint)`

Atomically subtracts the value in the second argument from the value pointed to by the first argument and returns the result.

The result is also assigned to the location pointed to by the first argument.

Full sequential consistency is guaranteed.

8.3.9 atomic_mul_u

Signature `uint atomic_mul_u(uint*, uint)`

Atomically multiplies the value pointed to by the first argument with the value in the second argument and returns the result.

The result is also assigned to the location pointed to by the first argument.

Full sequential consistency is guaranteed.

8.3.10 atomic_div_u

Signature `uint atomic_div_u(uint*, uint)`

Atomically divides the value pointed to by the first argument with the value in the second argument and returns the result.

The result is also assigned to the location pointed to by the first argument.

Full sequential consistency is guaranteed.

8.3.11 atomic_rem_u

Signature `uint atomic_rem_u(uint*, uint)`

Atomically computes the remainder from dividing the value pointed to by the first argument by the value in the second argument and returns the result.

The result is also assigned to the location pointed to by the first argument.

Full sequential consistency is guaranteed.

8.3.12 atomic_and_u

Signature `uint atomic_and_u(uint*, uint)`

Aotmically computes bit-wise AND between the value pointed to by the first argument and the value in the second argument and return the result.

The result is also assigned to the location pointed to by the first argument.

Full sequential consistency is guaranteed.

8.3.13 atomic_or_u

Signature `uint atomic_or_u(uint*, uint)`

Aotmically computes bit-wise OR between the value pointed to by the first argument and the value in the second argument and return the result.

The result is also assigned to the location pointed to by the first argument.

8.3.14 atomic_xor_u

Signature `uint atomic_xor_u(uint*, uint)`

Aotmically computes bit-wise XOR between the value pointed to by the first argument and the value in the second argument and return the result.

The result is also assigned to the location pointed to by the first argument.

Full sequential consistency is guaranteed.

8.3.15 atomic_load_s

Signature `int atomic_load_s(int*)`

Atomically loads the value from the memory location pointed to by the first argument.

Full sequential consistency is guaranteed.

8.3.16 atomic_store_s

Signature `void atomic_store_s(int*, int)`

Atomically sets the location pointed to by the first argument to the value in the second argument.

Full sequential consistency is guaranteed.

8.3.17 atomic_exchange_s

Signature `int atomic_exchange_s(int*.  int, int)`

Stores the value in the third argument to the location pointed to by the first argument if the value pointed to by the first argument is equal to the second argument. All of this happens atomically.

Returns 1 if the store happened; otherwise, returns 0.

Full sequential consistency is guaranteed.

8.3.18 atomic_add_s

Signature `int atomic_add_s(int*, int)`

Atomically adds the value in the second argument to the value pointed to by the first argument and returns the result.

The result is also assigned to the location pointed to by the first argument.

Full sequential consistency is guaranteed.

8.3.19 atomic_sub_s

Signature `int atomic_sub_s(int*, int)`

Atomically subtracts the value in the second argument from the value pointed to by the first argument and returns the result.

The result is also assigned to the location pointed to by the first argument.

Full sequential consistency is guaranteed.

8.3.20 atomic_mul_s

Signature `int atomic_mul_s(int*, int)`

Atomically multiplies the value pointed to by the first argument with the value in the second argument and returns the result.

The result is also assigned to the location pointed to by the first argument.

Full sequential consistency is guaranteed.

8.3.21 atomic_div_s

Signature `int atomic_div_s(int*, int)`

Atomically divides the value pointed to by the first argument with the value in the second argument and returns the result.

The result is also assigned to the location pointed to by the first argument.

Full sequential consistency is guaranteed.

8.3.22 atomic_rem_s

Signature `int atomic_rem_s(int*, int)`

Atomically computes the remainder from dividing the value pointed to by the first argument by the value in the second argument and returns the result.

The result is also assigned to the location pointed to by the first argument.

Full sequential consistency is guaranteed.

8.3.23 atomic_and_s

Signature `int atomic_and_s(int*, int)`

Aotmically computes bit-wise AND between the value pointed to by the first argument and the value in the second argument and return the result.

The result is also assigned to the location pointed to by the first argument.

Full sequential consistency is guaranteed.

8.3.24 atomic_or_s

Signature `int atomic_or_s(int*, int)`

Aotmically computes bit-wise OR between the value pointed to by the first argument and the value in the second argument and return the result.

The result is also assigned to the location pointed to by the first argument.

Full sequential consistency is guaranteed.

8.3.25 atomic_xor_s

Signature `int atomic_xor_s(int*, int)`

Aotmically computes bit-wise XOR between the value pointed to by the first argument and the value in the second argument and return the result.

The result is also assigned to the location pointed to by the first argument.

Full sequential consistency is guaranteed.

8.4 Memory management

8.4.1 gc_collect

Signature `void gc_collect()`

Instructs the GC to perform a full collection. This may cause a stop of the world.

8.4.2 gc_minimize

Signature `void gc_minimize()`

Instructs the GC to do minimal GC work. This function is appropriate for tight loops, and is relatively cheap.

8.4.3 gc_get_collections

Signature `uint64 gc_get_collections()`

Gets a value indicating the amount of collections the GC has performed.

8.4.4 gc_add_pressure

Signature `void gc_add_pressure(uint)`

Informs the GC that a significant amount of unmanaged memory (given by the argument) is about to be allocated.

8.4.5 gc_remove_pressure

Signature `void gc_remove_pressure(uint)`

Informs the GC that a significant amount of unmanaged memory (given by the argument) is about to be freed.

8.4.6 gc_is_generational

Signature `uint gc_is_generational()`

Gets a value indicating whether the GC is generational.

8.4.7 gc_get_generations

Signature `uint gc_get_generations()`

Gets the amount of generations managed by the GC. This is guaranteed to be a constant number.

Calling this function if the GC is not generational results in undefined behavior.

8.4.8 gc_generation_collect

Signature `void gc_generation_collect(uint)`

Instructs the GC generation given by the ID in the argument to perform a full collection. This may cause a stop of the world.

Calling this function if the GC is not generational results in undefined behavior.

8.4.9 gc_generation_minimize

Signature `void gc_generation_minimize(uint)`

Instructs the GC generation given by the ID in the argument to perform as much cleanup work as it can without stopping the world.

Calling this function if the GC is not generational results in undefined behavior.

8.4.10 gc_generation_get_collections

Signature `uint gc_generation_get_collections(uint)`

Gets a value indicating the amount of collections the GC has performed in the generation given by the ID in the argument.

Calling this function if the GC is not generational results in undefined behavior.

8.4.11 gc_is_interactive

Signature `uint gc_is_interactive()`

Gets a value indicating whether the GC is interactive (i.e. supports allocate and free callbacks). Returns 1 if the GC is interactive; otherwise, returns 0.

8.4.12 gc_add_allocate_callback

Signature `void gc_add_allocate_callback(void(Object&) cdecl)`

Adds a callback to the GC which will be called on every allocation made in the program. The parameter given to the function pointer is the newly allocated object. Note that the callback will be triggered right after the memory has been allocated.

Calling this function if the GC is not interactive or with a null callback pointer results in undefined behavior.

8.4.13 gc_remove_allocate_callback

Signature `void gc_remove_allocate_callback(void(Object&) cdecl)`

Removes a callback previously added with gc_add_allocate_callback. If the given callback was not registered previously, nothing happens.

Calling this function if the GC is not interactive or with a null callback pointer results in undefined behavior.

8.4.14 gc_set_free_callback

Signature `void gc_set_free_callback(Object&, void(Object&) cdecl)`

Adds a callback to the GC which will be called on the given object when it is no longer reachable (i.e. considered garbage). Note that this callback will be triggered just before the memory is actually freed. Passing a null value as the second argument will remove any existing callback for the given object. Passing any other value when a callback is already registered simply overwrites the existing callback.

The callback is automatically removed when the object is freed.

Calling this function if the GC is not interactive or with a null object results in undefined behavior.

8.4.15 gc_wait_for_free_callbacks

Signature `void gc_wait_for_free_callbacks()`

Blocks the current thread until all free callbacks that are currently enqueued have been processed by the finalization thread.

8.4.16 gc_is_atomic

Signature `uint gc_is_atomic()`

Gets a value indicating whether the GC is atomic (i.e. requires read or write barriers). Returns 1 if the GC is atomic; otherwise, returns 0.

8.4.17 gc_get_barriers

Signature `uint16 gc_get_barriers()`

Returns flags indicating which barriers the current GC requires.

Possible flags:

0x0	No barriers are required.
0x1	Read barriers are required for memory loads.
0x2	Write barriers are required for memory stores.

8.5 Math and IEEE 754 operations

8.5.1 nan_with_payload_f32

Signature `float32 nan_with_payload_f32(uint32)`

Produces a NaN (not a number) value with a given user payload. This abuses an obscure feature of IEEE 754 that allows 22 bits of a NaN value to be set to a user-specified value. This does of course mean that only 22 bits of the given payload will be inserted in the NaN value.

8.5.2 nan_with_payload_f64

Signature `float64 nan_with_payload_f64(uint64)`

Produces a NaN (not a number) value with a given user payload. This abuses an obscure feature of IEEE 754 that allows 51 bits of a NaN value to be set to a user-specified value. This does of course mean that only 51 bits of the given payload will be inserted in the NaN value.

8.5.3 nan_get_payload_f32

Signature `uint32 nan_get_payload_f32(float32)`

Extracts the 22-bit payload stored in a NaN (not a number) value.

8.5.4 nan_get_payload_f64

Signature `uint64 nan_get_payload_f64(float64)`

Extracts the 51-bit payload stored in a NaN (not a number) value.

8.5.5 is_nan_f32

Signature `uint is_nan_f32(float32)`

Returns 1 if the given value is NaN (not a number); otherwise, returns 0. This function is payload-aware, so NaNs with payloads will correctly be regarded NaN.

8.5.6 is_nan_f64

Signature `uint is_nan_f64(float64)`

Returns 1 if the given value is NaN (not a number); otherwise, returns 0. This function is payload-aware, so NaNs with payloads will correctly be regarded NaN.

8.5.7 is_inf_f32

Signature `uint is_inf_f32(float32)`

Returns 1 if the given value is positive or negative infinity; otherwise, returns 0.

8.5.8 is_inf_f64

Signature `uint is_inf_f64(float64)`

Returns 1 if the given value is positive or negative infinity; otherwise, returns 0.

8.6 Weak references

8.6.1 create_weak

Signature `Weak& create_weak(Object&)`

Creates a weak reference to an object given in the first parameter. Calling this function with a null parameter results in undefined behavior.

This function returns null if insufficient memory is available. The weak reference returned by this intrinsic must not be freed with `mem.free` or any other deallocation mechanism.

8.6.2 get_weak_target

Signature `Object& get_weak_target(Weak&)`

Gets the target of a given weak reference. Calling this function with a null weak reference results in undefined behavior.

The returned object may be null, since the target of the weak reference could have been collected since it was set.

8.6.3 set_weak_target

Signature `void set_weak_target(Weak&, Object&)`

Sets the target of a given weak reference. Calling this function with a null weak reference results in undefined behavior.

CHAPTER

NINE

CONCURRENCY

This document outlines the MCI's take on concurrency (atomicity, threading, and so on) during code execution.

9.1 General guarantees

The virtual machine generally doesn't make many guarantees in a concurrent environment. In general, managed code should not depend on atomicity guarantees made by the underlying hardware, as this makes code unportable in very subtle and hard-to-detect ways.

In other words, we do not guarantee that reads and writes of word-sized values will be atomic, as many other virtual machines do. While, in practice, you may find that they actually are (due to how the hardware works), it is not something we guarantee, and MCI will not consider atomicity of such operations when reordering instructions and performing other such optimizations.

The one thing that the virtual machine does guarantee is the consistency of reference values (this includes array and vector references). What this means is that dereferencing a reference (or an array/vector) will never result in an invalid memory access due to concurrency (save for the null case, naturally). Note that this is only guaranteed for references that are correctly aligned on a native word-size boundary (which is required for references to work correctly either way).

9.2 Atomic intrinsics

The virtual machine provides a number of intrinsics to do various atomic operations on word-size values (i.e. `int` and `uint`). Most basic arithmetic and logic operations are supported, simple loads and stores, as well as the CAS (compare-and-swap) operation.

The reason for not supporting fixed-size integer types is that implementing atomic operations for these across all supported architectures is hard, and may in some cases result in very inefficient code.

These intrinsics guarantee full sequencing (acquire/release semantics) on all supported architectures.

9.3 Threading

In general, the virtual machine relies heavily on the D runtime for its threading infrastructure. This is because the D runtime provides machinery to suspend and resume all threads for garbage collection runs (only relevant for stop-the-world GCs), and also provides a cross-platform TLS (thread-local storage) mechanism.

All threads that somehow execute managed code (be it via the interpreter by executing JIT-emitted code, or by calling through a trampoline) must therefore be attached to the runtime. There are some other subtle details like running D

module TLS constructors, attaching to the current garbage collector, and invoking managed thread entry points as well.

All trampolines generated by the virtual machine (via the `tramp` instruction) contain code to do all of the above. However, if threads jump directly into JIT-emitted code (this should by all means be avoided), they will have to do the attachment sequence manually before entering the managed code area.

All threads created through the intrinsic threading API are implicitly registered with the D runtime, hooked up to the garbage collector, etc. For such threads, this entire section can be ignored.

9.3.1 Termination

It is worth noting that once the entry point function in the program returns, the virtual machine will wait for all intrinsic threads that aren't daemon threads to join. This does not include threads created outside of the virtual machine. As such, it is the programmer's responsibility to ensure that threads outside of the virtual machine do not call into managed code once the entry point function has returned and the virtual machine has been shut down.

Intrinsic daemon threads will be forcefully terminated by the virtual machine. It is important that such threads can cope with this, and that they do not rely on any termination code to run.

CHAPTER

TEN

GARBAGE COLLECTION

This page details the standardized garbage collection infrastructure that the MCI provides to all programs running under the virtual machine.

10.1 Memory layout

All managed objects follow well-defined rules for physical layout of their contents.

All objects start with an object header. After the header comes the contents of the object. If the object is an array, the first thing after the header will be the size field, which is exactly one machine word long. After that comes whatever padding is needed to align elements to the native SIMD boundary. Following the padding are elements of the array, laid out contiguously. For vectors, the layout is exactly the same, except for the lack of a size field since the size is statically known (in other words, the padding space will likely be larger for vectors on some platforms). For plain structure objects, the fields follow immediately after the header.

10.1.1 Object headers

All managed objects contain a header that is exactly three machine words long. This header contains type information, garbage collector bits, and the field for user header data.

The specific layout is as follows:

Offset (32-bit/64-bit)	Description
0/0	Contains the type information pointer.
4/8	Contains garbage collection bits (meaning specific to GC implementation).
8/16	Contains the user data reference.

The type information pointer points to a structure that has a pointer to the actual `Type` object, a cached size, and a computed reference layout bitmap.

Generally, the raw header is not accessible to managed code at all. Reliance on the layout described here should be avoided except when consuming managed objects in native code.

10.1.2 Reference bitmaps

Most of the GC implementations use so-called reference layout bitmaps. These are very compact descriptions of where in a managed object references might be located. This information is useful to facilitate precise heap scanning.

Consider a type like this:

```
type Foo
{
    field Bar& bar;
    field int32 i;
    field float64 f;
    field Baz[] baz;
}
```

From this definition, it is clear that we do not need to scan the memory area consisting of `i` and `f` since they will never hold managed references. We encode this information in a bitmap where each bit represents a word of the type's memory layout. A 1 indicates that the word may hold a managed reference if non-null, while 0 indicates that it is just plain data.

The bitmap for `Foo` as defined above would be, on a 32-bit system:

```
00110001
```

On a 64-bit system:

```
0011001
```

The first three bits are always 001 because they represent the object header as described earlier. In the header, only the third field may hold a managed reference. After the header comes the `bar` field which is clearly managed. Next, we have two fields of plain data. Here is where the bitmap will differ depending on bitness; on a 32-bit system, there will be three words between `bar` and `baz` - one for `i` and two for `f`, while on a 64-bit system, there will only be two words - one for `i` and one for `f`. Due to alignment, an extra 4 bytes are added after `i`. Lastly, we have `baz` which is also clearly a managed reference.

The bitmap scheme works well regardless of the specific alignment imposed on a type by the programmer because references are required to always sit on word boundaries for correctness.

Note that the bitmap scheme is not currently used for arrays and vectors. In practice, this only matters for conservative GCs (they may pick up false pointers in arrays and vectors).

10.2 Reachability

An object is considered garbage when it is no longer reachable, directly or indirectly, from any GC roots (this includes stacks and registers). In the heap (that is, inside allocated objects), only direct pointers to other objects are considered. In roots, interior pointers are allowed (this is to facilitate passing object fields by reference).

Note that some garbage collectors may support interior pointers in the heap. However, this is a special case and is not a guaranteed feature of garbage collectors. It typically requires the collector to be completely conservative, which is highly undesirable.

10.2.1 Roots and ranges

Roots are single-word slots where the GC starts its scanning. A range is simply a contiguous sequence of such slots. Conceptually, all thread stacks are root ranges while global and TLS fields and machine registers are root slots. Root slots are required to be exactly one machine word because that's the size of a managed reference.

In addition to global and TLS fields, machine registers, and thread stacks, internal objects managed by the virtual machine may also be registered as roots.

10.2.2 Type precision

Since the MCI's type system is designed to fully support type-precise garbage collection, most GC implementations use some kind of type information to precisely identify managed references (typically bitmaps). This means that, for example, an integer cannot appear to be a valid managed reference and thus keep a managed object alive even though it is actually garbage.

Only the heap is scanned precisely in most GCs; roots and stacks are still scanned conservatively in all GCs. This may change in the future if we decide to compute precise stack maps, but this doesn't appear to be worth the effort and time/space cost currently.

10.2.3 Weak references

There is support for weak references in all garbage collectors the MCI provides. They are manipulated through the `create_weak`, `get_weak_target`, and `set_weak_target` intrinsics and are based on the `Weak` intrinsic type which is given special treatment by the virtual machine. The object a weak reference points to can be collected if there are no direct references to it other than through weak references. This can be useful for caching mechanisms in particular.

It is not actually guaranteed whether the target of a weak reference will be collected at all. Some garbage collectors may choose to treat weak references as strong references if absolutely necessary.

10.3 Compaction and copying

Garbage collectors may use so-called moving collection techniques. There are generally two forms of these: Compacting and copying. Both attempt to reduce heap fragmentation. Compaction does so by moving live objects while doing a collection. Copying uses two semispaces of equal size where live objects are copied to/from on each collection (this halves the heap space, but requires less passes over the heap than compaction).

The possible presence of these algorithms means that code must not assume that objects are fixed at a certain location in memory. The MCI's type system and ISA both try to enforce this by design (there are ways around this, but doing so is not supported in any way).

10.3.1 Pinning

The fact that objects may move arbitrarily means that native code can have trouble working with them. Since the MCI has no knowledge of external native code, it cannot correctly update references. The solution to this problem is called pinning: A pinned object cannot be collected. The MCI provides the `mem.pin` and `mem.unpin` instructions to do this.

Pinning of objects passed to `ffi` calls is required for correct results. This isn't statically verified, however, so undefined behavior can occur if pinning is not done (usually, this just results in bad memory accesses in the native code).

Practically, any object reachable directly from a root is pinned. However, this is not at all guaranteed, so pinning is still required for correct code.

It's important that objects be unpinned once pinning is no longer required. If an object is never unpinned, it will never be collected (until application shutdown).

10.4 Finalization

It is possible to register finalizers for all managed objects (including arrays and vectors). The `gc_set_free_callback` intrinsic registers a callback for a specific object. This callback will be called when the object is no longer reachable from any live object regardless of cycles (i.e. the finalizable object is reachable directly or indirectly from itself). Passing a null callback to `gc_add_free_callback` will remove any callback registered for the given object. Note that a callback is automatically removed before it is run.

No particular order of finalization is guaranteed. Callbacks should be programmed to not rely on order at all. Additionally, it is not guaranteed what thread a finalizer will run on, but it is guaranteed that the world will be resumed by the time a finalizer callback runs.

The `gc_wait_for_free_callbacks` intrinsic will block the calling thread until all queued finalization callbacks have been called. It can be useful if one needs to wait for a particular set of objects' finalization callbacks to run before continuing execution. Generally, this is achieved by letting those objects become garbage, calling `gc_collect`, and finally calling `gc_wait_for_free_callbacks`.

10.5 Barriers

Garbage collectors may require the use of read/write barriers. Contrary to what this terminology may suggest, barriers don't necessarily have anything to do with concurrency. They can be used for a wide array of things, and the specific purpose depends entirely on the GC implementation.

Barriers come in three flavors: Field reads/writes, array loads/stores, and indirect memory loads/stores. All of these barrier types are only called when managed types are involved. They are also only inserted into generated code if the GC specifically asks for them to be inserted, so there is no speed cost if a GC does not use barriers.

10.6 Garbage collectors

This section lists the current GC implementations available in the MCI.

10.6.1 D runtime garbage collector

GC name `dgc`

Type precision Conservative

Supports interior pointers Yes

Supports finalization No

Is generational No

Is incremental No

Is moving No

Uses barriers No

This GC uses the D runtime library's built-in garbage collector. It is entirely conservative and makes no use of type information. It has no support for finalization due to limitations in D's runtime library.

This GC is reasonably fast, but is geared towards native languages running in an uncooperative environment, and therefore doesn't make use of any of the information available for free in the MCI.

This GC supports interior pointers in the heap.

This is a stop-the-world collector with no support for parallel/concurrent GC.

10.6.2 Boehm-Demers-Weiser garbage collector

GC name `boehm`

Type precision Partially conservative

Supports interior pointers Partially

Supports finalization Yes

Is generational Optionally

Is incremental Optionally

Is moving No

Uses barriers No

This GC uses the Boehm-Demers-Weiser garbage collector (libgc). It has partial support for precise scanning using type bitmaps (only for structure types).

This GC supports interior pointers in the heap. However, in structure types (which use type bitmaps), they are only picked up when assigned to fields that are considered GC-managed (i.e. fields of reference, array, or vector types).

This GC is highly tuned through more than two centuries of development. It supports parallel marking and incremental collection.

This is a stop-the-world collector with no support for concurrent GC.

Note that this GC is not available on Windows. Also note that the MCI assumes that it is the only user of libgc in the process it's running in, so it will liberally set certain options without regarding any values they may have been set to previously (and also assumes those options won't be changed).

10.6.3 LibC garbage collector

GC name `libc`

Type precision N/A

Supports finalization Yes

Is generational No

Is incremental No

Is moving No

Uses barriers No

This GC performs no actual collection; it is equivalent to a null GC. It supports plain allocations and deallocations, and supports finalization (which is only triggered on explicit deallocation).

- *genindex*
- *search*