The Mu Micro VM Tutorial

Release 2.2

Kunshan Wang

Nov 23, 2017

Contents

This tutorial is about the Mu micro virtual machine.

A micro VM is a minimalist virtual machine as an underlying substrate for high-level programming language implementations. Mu is designed for languages that are concurrent, JIT compiled and use garbage collection.

CHAPTER 1

Language Implementor's Tutorial

This part is suitable for those who wish to implement a programming language on top of Mu.

1.1 Introduction

The Mu project addresses the difficulties in implementing managed languages, which include those languages that have a garbage collector and likely run on a virtual machine.

1.1.1 What Makes Language Implementation Difficult?

We identify three major concerns that make language implementation difficult, namely **concurrency**, **just-in-time compiling** and **garbage collection**.

Each of them is already difficult enough individually. But when handling all three at the same time, their combined complexity will multiply.

> For example, in a multi-threaded JIT-compiled garbage-collected program, when GC is triggered, the GC thread needs to ask all mutator threads running in JIT-compiled machine code to pause at the nearest GC-safe points (or "*yieldpoints*"), where stack maps are available to identify all references held by variables in the stack. In this task, the yieldpoints are inserted by the JIT-compiler, which is aware of GC. The stack maps are generated by the JIT-compiler, too. The handshake between mutators and the GC is very difficult to get right, too. Yieldpoints involve all three concerns mentioned above.

Because they are difficult, language implementations either omit them or implement them naively.

> Take CPython, the official Python implementation, as an example:
>
> - *lack of concurrency*: In CPython, there is a global interpreter lock (GIL) which must be obtained for any Python thread to run. This makes the execution of all multi-threaded Python programs sequential.
> - *lack of JIT-compiling*: CPython is interpreter-only and does not perform run-time optimisation based on JIT-compiling and type inference. A simple computation-intensive program may be up to 20

times slower than an equivalent C program. For comparison, PyPy, a Python implementation that does JIT-based optimisation, can run as fast as unoptimised C (GCC with -O0) in some cases.

- *naive garbage collector*: CPython uses a naive reference counting garbage collector. It increments or decrements the count every time a reference is created or destroyed. This naive algorithm is known to be more than 30% slower than a naive mark-sweep GC in total execution time.

In reality, **many languages designs are implementation driven**. Difficulties in implementation will result in **bad decisions being baked into the design** and hamper the evolution of the language in the future.

> Take PHP as an example. PHP was never supposed to be a programming language, but a set of pre-processing macros for personal home pages. As it evolves, it uses naive reference counting GC, copy-by-value semantic for arrays, and has reference types. An optimisation was also made to use copy-on-write to avoid copying. In 2002, a user found a problem involving arrays and references. Fives days later, the PHP developers decided that properly fixing the bug will put a considerably slowdown on the PHP performance and this behaviour is documented.
>
> Up till now in 2015, it has been 12 years since the problem was spotted. The problem can still be reproduced in today's PHP.

In conclusion, concurrency, JIT and GC are difficult. The difficulties lead to bad language designs and implementations, and should be abstracted out in a layer.

1.1.2 Why Micro Virtual Machines?

There are basically two ways to implement a managed language.

1. Building a VM from scratch. This is difficult because the developers have to address all of the three concerns. In reality, there are many successful projects taking this approach, such as PyPy, HHVM, V8 and so on. Most don't address all three concerns, and no code is shared between those projects.
2. Targeting an existing virtual machine, such as the JVM or the .NET CLR, which has high-quality implementations. The problems are *semantic gaps* and *huge dependencies*. Existing VMs are designed for particular kinds of languages. For example, the JVM is designed for static object-oriented languages. Optimisations usually done by JVM do not work for dynamic languages like Python. Jython, for example, still performs in the same order of magnitude as CPython. Moreover, using JVM introduces many unnecessary dependencies on Java-related packages.

Both approaches use **monolithic virtual machines**. Each such VM handles all aspects of the runtime of the language. For example, the JVM handles concurrency, JIT and GC, but also class loading, run-time type information, object-oriented programming (including virtual methods), and it comes with a comprehensive Java standard library. Since the VM is huge, it is difficult to build from scratch. Since it is monolithic, it is difficult to reuse its parts for other languages.

We propose an alternative concept: **micro virtual machines**. Analogous to *microkernels* in the operating system literature, a micro virtual machine only contains the parts that are absolutely necessary to be handled in the core of the VM. We suppose those parts are concurrency, JIT and GC. There is a separate program, called a **client**, sitting on the top of a micro virtual machine, interacting with the micro virtual machine and handling other (mostly language-specific) parts, in a similar fashion as *servers* interacting with the microkernel.

A micro virtual machine must be **low-level** and **minimal**.

- Being low-level means being close to the machine and thus minimise the semantic gap.
- Being minimal means it must push jobs that are not essential to the higher level, that is, the client. This is a separation of concern.
  - The micro virtual machine itself will be easier to design and implement. The minimalism also makes it practical to create a formally verified VM.

  - The client is not bound to the assumptions made by the low-level VM and can implement the language with maximum flexibility. The client has more responsibility, too, but it can still rely on the micro virtual machine for the three major concerns that are extremely difficult.

The minimalism pushes as much job as possible to the client side, potentially making the language implementer's job harder than targeting traditional VMs like JVM. To mitigate this limitation, **libraries** can be provided to assist the implementation of certain kinds of languages, like dynamic languages, functional languages, object-oriented languages and so on. Libraries are *not* part of the micro virtual machine. The library can be a framework or a package that is part of the client; it can also be pre-written code snippets.

The **Mu** project is a concrete micro virtual machine, in the same way seL4 is a concrete microkernel.

1.1.3 Separating the Specification and the Implementation

Mu separates its specification and its implementations, making it possible to create many different implementations for different purposes. In theory, there may be a simple proof-of-concept implementation, a high-performance implementation for productional use, an extensible modular implementation for researching, a formally verified implementation for highly-assured applications, and so on.

The Mu specification defines the behaviour and the interface of Mu.

Currently, Mu has two implementations:

- Holstein, the reference implementation, is a **simple** Mu implementation and allow early evaluators to do experiment with the interface of Mu. The simplicity allows Mu reference implementation to be agilely changed when the specification changes. It is **not** a high-performance implementation.
- Zebu, the fast implementation, is developed from the first day to be **fast**. It is written in Rust, and has a optimizing compiler and a high-performance garbage collector. It is currently implemented as an ahead-of-time compiler. Some functionalities are still in development.

1.2 Getting Started

1.2.1 Where is Everything?

The specification is the canonical source of information about the implementation-neutral parts of Mu.

This tutorial will use Holstein, the current reference implementation. Keep in mind that it is not high-performance. It is interpreted and single-threaded, does excessive checking and is very slow.

1.2.2 Get Mu

Go to https://gitlab.anu.edu.au/mu/mu-impl-ref2 and follow the README file. That repository also contains a sample factorial program compiled from an RPython client and a loader to run it. Read the README file.

1.2.3 Pick your language

The client of the reference implementation can be written in Scala or in C. If you use Scala, you can just use the classes in the `microvm-refimpl2` repository since the reference implementation is already written in Scala. If you use C, there is a C binding in the `cbinding` directory. Read the README file for instructions. There is also a sample client program `test_client`.

The following part of the tutorial assume you use Scala. Don't worry if you use C because the interface is quite similar.

1.3 Basic Interaction

You are writing a Mu *client*, which controls the Mu micro VM via the Mu client interface, usually simply called "the API".

In Scala, the class `uvm.refimpl.MicroVM`, which implements the `MuVM` struct in the specification, represents a micro VM instance. You can start Mu by creating a `MicroVM` instance:

```
val microVM = MicroVM()
```

Just as simple as this. The instance also internally allocates memory for its heap. The default heap size is quite big, and is usually enough for experiment.

1.3.1 Mu Contexts

The client more often interacts with the micro VM via "Mu context". They are called "MuCtx" in the spec, and `uvm.refimpl.MuCtx` in Holstein. A MuCtx is a context created from a MicroVM instance. It can hold Mu values for the client, access the Mu memory, load bundles and let the client perform many tasks on Mu.

Why not directly do these things on the MicroVM instance? Why add another layer? There are two reasons.

Reason 1: because Mu is designed to be **multi-threaded**. By design, multiple client threads can interact with the Mu micro VM concurrently.

Caution: In fact, unfortunately, the *reference implementation* is based on a single-thread interpreter. The program itself is **not thread safe**. Do not run more than one *client threads* in the reference implementation.

But Mu is designed for multi-threaded envirionments. The limitation in the implementation does not change the fact that Mu and its clients need to think in a multi-threaded way. A carefully designed interface will eventually allow a more efficient implementation.

Despite this limitation, the reference implementation can still run multiple **Mu threads**. Mu threads are interpreted one instruction for each thread on a round robin scheduler.

If multiple client threads were accessing the single MicroVM object concurrently, synchronisation (such as locks) must be employed to guarantee thread safety. This is where the "context" comes in. In Mu's design, MuCtx instances are not allowed to be used by two threads concurrently, thus avoided many cases where synchronisation were necessary. For example, accessing the garbage-collected Mu memory via MuCtx does not need locking. A MuCtx instance can also hold a thread-local memory pool so that a client thread can allocate memory in the garbage-collected Mu heap without having to deal with the shared global memory pool every time unless the local pool is exhausted.

Reason 2: because the *type system* of Mu is usually very different from the client's. Notably, the Mu's type system contain **object reference** types which points to the Mu heap and **must be traced by the garbage collector**. MuCtx can hold Mu references for the client so that whenever garbage collection happens, it always knows what objects are still kept alive by a reference held for the client.

If you used JNI before, you may find this design familiar. In fact, this design is inspired by the JNI. JNI uses opaque "handles" to refer to Java object, so does the Mu API. However, for performance reason, opaque handles are not the only way to expose garbage-collected Mu objects to *native* programs. Mu has a more efficient but unsafe native interface which supports "object pinning". That is an advanced topic.

Note: There is a difference between a **Mu client** and a **native program**.

A Mu client is a program that controls the Mu micro VM. In theory, a Mu client can be written in any languages, from C to Scala, Python, JavaScript, etc. The Mu API is the interface between the *client* and the *micro VM*. It includes the IR and the API, and the purpose is to control the micro VM.

A native program is a program that does not run in the Mu micro VM, and is usually written in C or other unmanaged languages. libc is one such example. The native interface involves pointer-based raw memory access and calling conventions that allow Mu programs to call C programs and vice versa in some specific ways. The main purpose is to make system calls (obviously a VM that cannot `read` or `write` is almost useless), and to interact with programs that do not run on Mu, including C programs and those written in other languages.

Using Mu Contexts

You can create a `MuCtx` instance by invoking the `newContext()` method on the `MicroVM` instance:

```
val ctx = microVM.newContext()
```

and you need to close the context in order to release the resources it is holding inside:

```
ctx.closeContext()
```

The API, i.e. the methods of `MicroVM` and `MuCtx`, are defined by the API chapter of the specification. The scala binding matches the spec.

Let's see what MuCtx can do. You don't need to understand all of them now, since they will be covered in more depth in later chapters.

Create contexts from `MicroVM` instance:

```
val microVM = MicroVM()
val ctx = microVM.newContext()
```

Holstein can load Mu bundles from the text form. This method of loading bundle is deprecated in favour for the IR building API which avoids the text-form IR parser.

```
ctx.loadBundle("""
    .typedef @i64 = int<64>
    // more Mu IR code here
""")
```

`MuCtx` can hold Mu values for the client. Mu values have a specific int size.

```
val handle1 = ctx.handleFromInt(0x123456789abcdef0L, 64)
val handle2 = ctx.handleFromInt(0x12345678L, 32)
val handle3 = ctx.handleFromInt(0x1234L, 16)
val handle4 = ctx.handleFromDouble(3.14)
val handle5 = ctx.handleFromPtr(ctx.idOf("@someType"), 0x7fff0000018L)
```

It can allocate objects in the Mu heap. The handle is held in ctx so that GC can find all of them.

```
val handle6 = ctx.newFixed(ctx.idOf("@someType"))
```

It can create Mu stacks and Mu threads

```
val hFunc   = ctx.handleFromFunc(ctx.idOf("@some_function"))
val hStack  = ctx.newStack(hFunc)
val hArg0   = ....
val hArg1   = ....
val hArg2   = ....
val hThread = ctx.newThread(hFunc, PassValues(Seq(hArg0, hArg1, hArg2)))
```

It can access the Mu memory

```
val hObjRef = ctx.newFixed(ctx.idOf("@int_of_64_bits"))
val hIRef   = ctx.getIRef(hObjRef)
val hValue  = ctx.load(MemoryOrder.SEQ_CST, hIRef)
```

It can introspect the stack states

```
val hStack2 = .....
val hCursor = ctx.newCursor(hStack2)
val funcID  = ctx.curFunc(hCursor)          // function ID
val hVars   = ctx.dumpKeepalives(hCursor)   // local variables
```

It can modify the stack states (a.k.a. on-stack replacement, OSR)

```
ctx.nextFrame(hCursor)
ctx.popFramesTo(hCursor)
val hFunc2  = ctx.handleFromFunc(...)
ctx.pushFrame(hFunc2)
```

1.3.2 Threads and Stacks

Mu programs are executed on Mu threads. A thread is the unit of CPU scheduling, and Mu threads are usually implemented mirroring operating system threads. Multiple Mu threads may execute concurrently.

Each Mu thread runs on a Mu **stack**. A stack, commonly known as a *control stack*, is the state of execution, represented in Mu as a list of *frames*. Each frame corresponds to a Mu function version, and records which instruction should be executed next and what are the values of local variables.

Mu clearly distinguish between threads and stacks. If you used traditional thread APIs, such as the Java or the PThread API, you may already have the mental model that "a thread has a stack, which has many frames, so threads and stacks are interchangeable". But in Mu, the relation of stacks and threads is much more flexible. A thread can stop executing on one stack and resume another stack, which gives "coroutine" behaviours. Multiple threads can also share a much bigger stack pool and implement the M*N threading model.

In order to start executing a Mu program, the client should create a Mu stack and a Mu thread. In order to stop executing, the Mu thread should execute the `@uvm.thread_exit` instruction.

1.3.3 Trap Handling

There is one special instruction, `TRAP`, that needs special attention since the beginning. During the execution of Mu programs, if a Mu thread executes a `TRAP` instruction, the thread temporarily detaches from its stack and gives control back to the client. At any moment, there is one trap handler registered in a Mu instance. A trap handler is a client function that will be called whenever a `TRAP` instruction is executed. The trap handler gains access to the thread and the stack that caused the `TRAP`.

Using the API, the client can to introspect the execution state of each of its frames, see the values of local variables, and even replace existing frames with new frames for new functions (this is called *on-stack replacement*, or OSR).

The trap handler is a great opportunity for the client to do many things. The clever placement of `TRAP` instructions and the implementation of the trap handler is key to a good language implementation. Traps can be placed after sufficient run-time statistics are collected so that the client can optimise the program. Traps can also be used for lazy code loading, de-optimising speculatively generated code, and debugging.

Scala API

The trap handler is registered by the `setTrapHandler` method on the `MicroVM` instance.

```
microVM.setTrapHandler(theTrapHandler)
```

The trap handler is an instance of the `uvm.refimpl.TrapHandler` trait.

```
trait TrapHandler {
  def handleTrap(ctx: MuCtx, thread: MuThreadRefValue, stack: MuStackRefValue,␣
↪watchPointID: Int): TrapHandlerResult
}
```

A new `MuCtx` instance is created for this particular trap event. It is passed to the trap handler as the first argument `ctx`. The `thread` and the `stack` argument are handles of the thread that executed the `TRAP`, and the stack it was bound to, respectively. These two handles are held by `ctx`. The `watchPointID` argument is about "watch points", which will be discussed later.

You probably only need one trap handler per program, so it is recommended to register it after created the Mu instance.

Inside the trap handler, you can use any API functions.

The return value of the trap handler tells Mu "how the current thread should continue". There are three options:

- Terminate the current thread.
- Rebind the thread to a stack.
  - When rebinding, pass some values to the top frame and let it continue normally.
  - When rebinding, raise an exception and continue exceptionally.

When rebinding, the stack could be the previous stack, i.e. the `stack` argument of the trap handler, or a totally different stack. In the former case, it will continue after the `TRAP` instruction. In the latter case, the trap handler swaps the thread to a different stack, so the thread will continue in a totally different context.

The return type `uvm.refimpl.TrapHandlerResult` has several cases:

```
abstract class TrapHandlerResult
object TrapHandlerResult {
  case class ThreadExit() extends TrapHandlerResult
  case class Rebind(newStack: MuStackRefValue, htr: HowToResume) extends␣
↪TrapHandlerResult
}

abstract class HowToResume
object HowToResume {
  case class PassValues(values: Seq[MuValue]) extends HowToResume
  case class ThrowExc(exc: MuRefValue) extends HowToResume
}
```

So you can return one of the cases from the trap handler:

```
// Just for convenience
import TrapHandlerResult._
import HowToResume._

```

```
// Terminate the thread.
return ThreadExit()

// Rebind to the old stack, pass some values and contunue normally
// Assume "stack" is the argument of the handleTrap method
val v1 = ctx.handleFrom......(...)
val v2 = ctx.handleFrom......(...)
val v3 = ctx.handleFrom......(...)
return Rebind(stack, PassValues(Seq(v1, v2, v3)))

// Rebind to the old stack, pass an empty list of values and contunue normally
// Assume "stack" is the argument of the handleTrap method
return Rebind(stack, PassValues(Seq()))

// Rebind to the old stack, throw an exception.
// Assume "stack" is the argument of the handleTrap method
// In Mu, an exception is just an object reference.
val e = ctx.newFixed(......)
return Rebind(stack, ThrowExc(e))

// Rebind to a different stack, passing 0 values.
val func = ctx.handleFromFunc(...)
val stack2 = ctx.newStack(func)
return Rebind(stack2, PassValues(Seq()))
```

C API

In the C API, you should use `mvm->set_trap_handler(mvm, handler, user_data)` to register the trap handler.

The signature of the trap handler is a bit complicated:

```
typedef void (*MuTrapHandler)(MuCtx *ctx, MuThreadRefValue thread,
    MuStackRefValue stack, int wpid, MuTrapHandlerResult *result,
    MuStackRefValue *new_stack, MuValue **values, int *nvalues,
    MuValuesFreer *freer, MuCPtr *freerdata, MuRefValue *exception,
    MuCPtr userdata);
```

The first four arguments `ctx`, `thread`, `stack` and `wpid` are the same as Scala. The next seven arguments `result`, `new_stack`, `values`, `nvalues`, `freer`, `freedata` and `exception` are output arguments. They allow the C program to encode the counterpart of `TrapHandlerResult`. Since the client in C needs to pass an array of values to Mu, it also needs to tell Mu how to de-allocate that array because there is not a standard way to de-allocate C objects. The `userdata` is an arbitrary pointer the client provided when registering the trap handler. This allows the trap handler to depend on extra client-decided contexts, because C does not have closures.

See the trap handling section of the Mu Specification for more information about trap handling in C.

1.3.4 Working Example

The following Scala program will create a Mu micro VM and execute a simple Mu IR program. You can ignore details of the Mu IR now (except the TRAP instruction), but instead focus on the interaction between the client, the Mu thread, and the trap handler.

The order of execution is labelled from `#1` to `#26`.

```scala
package tutorial

import uvm.refimpl._

object Interact extends App {

  // Create the Mu instance
  val microVM = MicroVM()      // #1

  // Implicitly convert names to IDs
  implicit def idOf(name: String) = microVM.idOf(name)   // #2

  // Create the context
  val ctx = microVM.newContext()   // #3

  ctx.loadBundle("""               // #4
.typedef @i64 = int<64>

.const @I64_1 <@i64> = 1

.funcsig @main.sig = (@i64) -> ()

.funcdef @main VERSION %v1 <@main.sig> {     // #12
  %entry(<@i64> %n):
    %n2 = ADD <@i64> %n @I64_1               // #13
    [%trap] TRAP <> KEEPALIVE (%n2)          // #14
    COMMINST @uvm.thread_exit                // #21
}
""")

  // Create the trap handler
  val myTrapHandler = new TrapHandler {
    def handleTrap(ctx: MuCtx, thread: MuThreadRefValue,
        stack: MuStackRefValue, watchPointID: Int): TrapHandlerResult = {

      // Create a cursor to introspect the stack
      val cursor = ctx.newCursor(stack)         // #15
      val curInstID = ctx.curInst(cursor)       // #16

      ctx.nameOf(curInstID) match {
        case "@main.v1.entry.trap" => {         // #17
          // Dump the keep-alive variables
          val Seq(n2: MuIntValue) = ctx.dumpKeepalives(cursor)     // #18
          ctx.closeCursor(cursor)

          // Print the value
          val n2Int = ctx.handleToSInt(n2)             // #19
          printf("The value of n2 is %d.\n", n2Int)

          // Return to Mu from the trap handler
          TrapHandlerResult.Rebind(stack, HowToResume.PassValues(Seq()))   // #20
        }
      }
    }
  }

  // Set the trap handler
  microVM.setTrapHandler(myTrapHandler)            // #5
```

```

    // Create the stack and the thread
    val mainFunc = ctx.handleFromFunc("@main")       // #6
    val st = ctx.newStack(mainFunc)                  // #7

    val fortyTwo = ctx.handleFromInt(42, 64)         // #8

    val th = ctx.newThread(st, None, HowToResume.PassValues(Seq(fortyTwo)))  // #9

    // Close the context
    ctx.closeContext()     // #10

    // Let the reference implementation run
    microVM.execute()      // #11

    // #22
}
```

There are some key steps:

- `#1` and `#3` creates the Mu instance and a context, respectively.
- `#4` loads the initial bundle using the MuCtx. You can directly embed the text-form Mu IR in your source code.
- `#5` registers the trap handler. This handler will handle all traps in the future.
- Using the MuCtx, `#6-9` creates a Mu thread from a given Mu function, whose name happens to be `@main`. "main" is not a special name. When creating the thread, the initial argument, the 64-bit integer 42, is passed to the `@main` function. At `#10`, the MuCtx `ctx` is no longer useful and can be closed.
- Once created, the Mu thread can execute. But the reference implementation needs to call the `microVM.execute()` method `#11` to execute all Mu threads in the only Scala (JVM) thread.
- During the execution of the Mu thread, it hits the trap at line `#14`. The control then transfers to the trap handler. Note that the `KEEPALIVE` clause specifies which local variables are eligible for introspection. In this case, it is only `%n2`.
- In the trap handler, the value of client-specified local variables (keep-alive variables) are dumped at `#18` and printed. Since the previous `ADD` instruction `#13` adds one to the number, it should print "43" here.
- Then at `#20`, the trap handler re-binds the thread with the old stack, passing an empty list of values back to the TRAP.
- Then it continues from the Mu function after TRAP `#14`. The `thread_stop` instruction at `#21` stops the Mu thread.
- In the reference implementation, the `execute()` function at `#11` ends when the last Mu thread stopped. Then the example program quits. The specification does not specify how a Mu micro VM should end.

1.3.5 Summary

- A MicroVM instance is the heart of the Mu micro VM.
- The client interacts with the micro VM mostly via MuCtx. A context serves only one client thread. It holds Mu values, including garbage-collected object references.
- In Mu, threads and stacks are loosely coupled. Threads can swap from one stack to another.
- The `TRAP` instruction gives the control back to the client from an executing Mu thread.

- To start everything: create a MicroVM, create a MuCtx, load a bundle, create a stack and create a thread. The `MicroVM.execute()` API function is specific to the reference implementation.

1.4 Mu Intermediate Representation

The client generates code in the format of **Mu Intermediate Representation**, or **Mu IR** for short. The IR is the language in which programs are represented in the Mu micro VM. The Mu IR is defined by the Mu IR chapter of the Mu specification.

The structure of the Mu IR is an AST. Mu bundle has a text form for human readability. A bundle can also be built using the IR building API which calls into Mu to build an AST inside Mu. The IR building API is designed for productional setting. This tutorial will use the text-form API.

1.4.1 Bundle

The client submits one **bundle** (a piece of Mu IR code) at a time to the Mu micro VM.

```
ctx.loadBundle("""
    // insert your bundle here
""")
```

A bundle defines many *types*, *function signatures*, *constants*, *global cell* and *functions*. The client may submit multiple bundles one after another.

After submitting, Mu knows about those things. Types can be used, global cells are allocated, and functions are callable.

A bundle looks like this:

```
// Type
.typedef @i64 = int<64>

// Function signature
.funcsig @i_to_i = (@i64) -> (@i64)

// Constant
.const @I64_1 <@i64> = 1
.const @I64_2 <@i64> = 2

// Global cell
.global @g_foo <@i64>

// Function declaration (no body)
.funcdecl @factorial <@i_to_i>

// Function definition (with body)
.funcdef @fibonacci VERSION %v1 <@i_to_i> {
    %entry(<@i64> %n):                              // Basic block
        %lt = SLT <@i64> %n @I64_2                      // Instructions
        BRANCH2 %lt %small(%n) %big(%n)                 // Instructions

    %small(<@i64> %n):                              // Basic block
        RET %n                                          // Interaction

    %big(<@i64> %n):                                // Basic block
        %nm1 = SUB <@i64> %n @I64_1                     // Instruction
        %nm2 = SUB <@i64> %n @I64_2                // Instruction
        %v1  = CALL <@i_to_i> @fibonacci (%nm1)    // Instruction
        %v2  = CALL <@i_to_i> @fibonacci (%nm2)    // Instruction
        %rv  = ADD <@i64> %v1 %v2                  // Instruction
        RET %rv                                    // Instruction
}
```

If you have used LLVM before, the Mu IR is the counterpart of LLVM modules.

1.4.2 Names

You may have noticed that there are names for almost all entities. In the Mu IR, there are two kinds of names: global names and local names. Global names start with `@` and local names start with `%`. The allowed characters in names are `[a-zA-Z0-9_.]`.

In fact, local names are just syntax sugars of some global names, that is, anything that has a name has a global name. This will be discussed later.

At this moment, you only need to know what a bundle may contain. Their details will be discussed in the following chapters.

1.5 Type System

The Mu type system is defined in the specification.

Like many programming languages and frameworks, Mu also has a type system.

The Mu type system is low level. There is no object-oriented programming concepts, such as class, inheritance, polymorphism. There is no high-level concepts such as strings, either. The language implementer is responsible to implement these high-level concepts.

But the Mu type system is also not too low level. Notably, unlike C, C++ or LLVM, the Mu type system still has object reference types in it, and the garbage collector is fully aware of the presence of them. The main idea is, as long as you use the Mu type system, and refer to heap objects using references, you can forget about garbage collection details, such as stack maps, GC-safe points, and read/write barriers.

1.5.1 Overview

Some of the types (actually *type constructors*, explained later) contain angular brackets. These are parameters to these types which may be integer literals, other types or function signatures.

The types in the Mu type system can be put into several categories:

1. Scalar value types: `int<n>`, `float`, `double`, `uptr<T>` and `ufuncptr<sig>`. These types represent plain values.
2. Scalar reference types: `ref<T>`, `iref<T>`, `weakref<T>`, `funcref<sig>`, `threadref`, `stackref`, `framecursorref`, `irbuilderref` and `tagref64`. These types refer to "things" in the Mu micro VM. All such references are opaque in the sense that their representation is implementation dependent.
3. Composite types: `struct<F1 F2 ...>`, `array<T n>`, `hybrid<F1 F2 ... V>` and `vector<T n>`. These types combine simpler types into more complex types.
4. The void type `void`. It has only one use case: when used as the "referenced type" of references or pointers, it conveys the meaning of "reference/pointer to anything".

This is a complete list of Mu types. All values of Mu come from one of these types.

1.5.2 Define Types and Function Signatures

Type definition

To define a type in Mu, you use the top-level type definition: `.typedef`.

```
.typedef @i1     = int<1>
.typedef @i8     = int<8>
.typedef @i16    = int<16>
.typedef @i32    = int<32>
.typedef @i64    = int<64>
.typedef @float  = float
.typedef @double = double

.typedef @ptri32 = uptr<@i32>
.typedef @foo.fp = ufuncptr<@foo_sig>

.typedef @refi32  = ref <@i32>
.typedef @irefi64 = iref<@i64>
.typedef @weakreffloat = weakref<@float>

.funcsig @foo.sig = (@i32) -> (@i32)
.typedef @foo.fr  = funcref<@foo_sig>

.typedef @stackref  = stackref
.typedef @threadref = threadref
.typedef @stkref    = stackref
.typedef @thrref    = threadref
.typedef @fcref     = framecursorref
.typedef @ibref     = irbuilderref

.typedef @struct1 = struct<@i32 @i32 @i32>
.typedef @struct2 = struct<@i64 @double>
.typedef @array1  = array<@i32 10>
.typedef @array2  = array<@i32 4096>
.typedef @hybrid1 = hybrid<@i64 @i32>
.typedef @hybrid2 = hybrid<@i8>
.typedef @hybrid3 = hybrid<@i64 @i64 @i64 @float>
.typedef @4xi32   = vector<@i32 4>

.typedef @void = void
```

Every type defined in the Mu IR has a name, which is on the left side of the equal sign. All characters in `[a-zA-Z0-9_.]` are legal. You can use the dot `.` arbitrarily in the name. So the dot in `@foo.sig` does not mean anything special to Mu. On the right side of the equal sign is the **type constructor**: it constructs a type. Some type constructors take parameters while others do not.

What are type constructors?

If we imagine a Mu type as a Java or C++ object, then the type constructor is like the constructor of such an object. `int` is just the abstract concept of integer, but `int<32>` is a concrete 32-bit integer type. Similarly `ref<@i32>` constructs a reference type to `@i32`:

```
.typedef @i32    = int<32>
.typedef @refi32 = ref<@i32>
```

Some type constructors, such as `float`, `double`, `threadref` or `void`, do not take any parameters. You can consider them as C++/Java constructors with an empty parameter list. You may have written `new Object()` or `new StringBuilder()` before. Similarly you define a concrete instance of `float` type in this way:

```
.typedef @float    = float
.typedef @blahblah = float
```

, where the name `@float` or `@blahblah` are just names.

When types or function signatures are taken as argument, their names (such as `@i32`, `@float` and `@void`, not `int<32>`, `float` or `void`) are used. So the following are not accepted by Holstein:

```
.typedef @refi32  = ref<int<32>> // ERROR! int<32> must be defined separately.
.typedef @refvoid = ref<void>    // ERROR! void must be defined separately.
.typedef @bar.ref = funcref<(@i32) -> (@float)> // ERROR! The signature must be␣
↪defined separately.
```

But these are right:

```
.typedef @i32     = int<32>
.typedef @refi32  = ref<@i32>  // Correct.

.typedef @void    = void
.typedef @refvoid = ref<@void> // Correct.

.typedef @float   = float
.funcsig @bar.sig = (@i32) -> (@float)
.typedef @bar.ref = funcref<@bar.sig>   // Correct.
```

Note: So why does Mu force all types to be "constructed" at the top level? Well, that's what Holstein accepts now. There are alternative text Mu IR parsers that accept in-line types such as `ref<int<32>>`.

Actually, productional Mu and client implementations will use the IR building API. It will skip the text parsing phase completely.

The reason why Holstein was designed like that was to let the text match the actual data structure of the IR. In the IR building API, each type is a "node". Types that have parameters (such as `ref<T>`) refer to other nodes by their IDs. Similarly, in the text form, such type constructors refer to other types by names.

If you have used LLVM before, you may find that you can write types "directly", "inline", in the LLVM IR, such as:

```
%c = add i32 %a, %b
%f = fadd double %d, %e
%g = load i32* %x
```

But have a look at the C++ API of the LLVM:

```
Type *i32 = Type::getInt32Ty(ctx);
Type *i64 = IntegerType::get(ctx, 64);  // alternative method
Type *floatTy  = Type::getFloatTy(ctx);
Type *doubleTy = Type::getDoubleTy(ctx);
Type *voidTy   = Type::getVoidTy(ctx);

Type *blahblah = Type::getFloatTy(ctx);

Type *ptri32 = Type::getInt32PtrTy(ctx);
Type *ptri64 = PointerType::getUnqual(i64);
```

In this API, the programmer still needs to refer to types by pointers to the types. So this API is more similar to having to define (or, at least, make pointers to) the types separately.

On the other hand, there is only 19 types in the Mu type system, among which only 6 do not take arguments. Even if the client programmer has to define each and every types, all common types can be defined in about 20 lines as *above*, and his/her pain ends there.

Function signature definition

A **function signature** defines the parameter types and the return types of a function. It is defined by the `.funcsig` top-level definition:

```
.typedef @i32      = int<32>
.typedef @float    = float

.funcsig @sig1     = (@i32) -> (@float)
.funcsig @sig2     = (@i32 @i32 @i32) -> (@i32 @float)
.funcsig @sig3     = () -> (@i32)
.funcsig @sig4     = (@i32) -> ()
.funcsig @sig5     = () -> ()

.typedef @funcref1   = funcref <@sig1>
.typedef @ufuncptr1 = ufuncptr<@sig1>
```

On the left side of = is the name of the signature. On the right side is the function signature constructor. In Mu, a function takes 0 or more parameters and return 0 or more values. It is written in the form `(parameter types) -> (return types)`.

A function signature is **not** a type. Unlike the C or C++ programming language, there is no “function type” in Mu. In fact, in C, if an expression has function type, it is implicitly converted to the pointer of that function. Mu takes the explicit approach: there are two types that use function signatures:

- The `funcref<sig>` type refers to a Mu function which has signature `sig`.
- The `ufuncptr<sig>` type is a pointer that points to a native function that has signature `sig`.

When defining or declaring functions, such as:

```
.funcdecl @foo <@sig1>

.funcdef @bar VERSION %v1 <@sig2> {
    ...
    %rv = CALL <@sig1> @foo (...) // arguments omitted
    ...
}
```

The names of the functions `@foo` and `@bar` has the `funcref<@sig1>` and the `funcref<@sig2>` type, respectively, when used as a value.

1.5.3 Details

This section will only discuss the most important types. For more details, you can read the Type System section of the specification.

Integer and FP types

`int<n>` is the **integer** type of `n` bits. Like LLVM, the `int` type is fixed-length. For example, `int<32>` is the 32-bit integer type.

```
.typedef @i32 = int<32>
.typedef @i64 = int<64>
```

It is also signedness-neutral: whether an integer is signed or not depends on the operation, not the type. Most instructions, such as `ADD`, `SUB`, `MUL`, work correctly for both signed and unsigned integers. Some instructions have signed and unsigned variants, such as `SDIV`/`UDIV`, `FPTOSI`/`FPTOUI`.

Like LLVM, `int<1>` is returned by most instructions that return Boolean results, such as `EQ` and `SLT`.

```
.typedef @i1 = int<1>
```

`float` and `double` are the IEEE 754 single and double-precision **floating point** number types, respectively.

```
.typedef @float  = float
.typedef @double = double
```

Like LLVM but unlike some intermediate languages such as C minus minus, Mu does not use a single type (such as "bits32") to hold both integers and FP numbers, because in modern machines integers and FP numbers are usually held in different kinds of registers.

References to the memory

`ref<T>` is the **object reference** type. It refers to objects in the garbage-collected Mu heap.

`iref<T>` is the **internal reference** type. It refers to a *memory location*, that is, a place in the Mu memory that can be loaded or stored. A field of a heap object is a memory location.

Attention: "Memory location" does not mean "address". Do not assume a Mu heap object or any other memory locations have addresses. This is very important in Mu. This will discussed in details in later chapters. The specification contains some explanation

Both `ref` and `iref` may be `NULL`.

Note: Sorry for the billion-dollar mistake, but `NULL` is really easy to implement, and Mu is closer to the machine. The client, on the other hand, should implement a decent language and help the programmers prevent such mistakes.)

The `<T>` type parameter is the type of the heap object it refers to.

For `ref<T>`, the `T` means it refers to a heap object of type `T`. For example, `ref<@i32>` refers to a heap object of `@i32` type, which we previously defined as `int<32>`:

```
.typedef @refi32    = ref<@i32>
.typedef @refdouble = ref<@double>

.typedef @link = ref<@link>
```

In the last line, `@link` is recursively defined as `ref<@link>`. It means it refers to a heap object, whose entire content is an object reference to the same type, or `NULL`. It is very similar to the C definition: `struct Link { struct Link *next; }`. Mu does not need `struct` to construct recursive types.

For `iref<T>`, the `T` means it refers to a memory location of type `T`. So if you use the `LOAD` instruction on an `iref<@i32>`, you get a value of type `@i32`. You can also `STORE` an `@i64` value to a memory location referred by an `iref<@i64>`.

```
.typedef @irefi32   = iref<@i32>
.typedef @irefi64   = iref<@i64>
```

References to Mu functions

`funcref<sig>` is the **function reference** type. It refers to a Mu function. Whenever you call a Mu function, you call it with its function reference. `sig` is the function signature.

```
.funcsig @sig1      = (@i32) -> (@float)
.typedef @funcref1  = funcref <@sig1>
```

`funcref` only refers to Mu functions. It cannot refer to C functions (that is what `ufuncptr` is for).

Like other references, `funcref` can also be `NULL` (sorry).

Aggregate types

Among all aggregate types, `hybrid` is the only "variable-length" types. All others are "fixed-length".

Fixed-length aggregate types

`struct<F1 F2 ...>` is the **structure** type. Like the *struct* type in C, it has many fields of types `F1`, `F2`, ...

```
.typedef @struct1 = struct<@i32 @i32 @i32>
.typedef @struct2 = struct<@i64 @double>
```

Structs may contain other structs, arrays or vectors, but cannot contain themselves (otherwise it will be infinitely big). It must have at least one field. But it may contain references so that you can allocate many structs in the Mu heap, each refer to another object.

```
.typedef @ListNode    = struct<@i64 @ListNodeRef>
.typedef @ListNodeRef = ref<@ListNode>
```

`array<T n>` is the **fixed-size array** type. `T` is the element type. `n` is an integer literal and it is part of the type. A particular `array<T n>` holds exactly `n` instances of `T`. For example, `array<@i32 10>` contains exactly 10 `@i32` values:

```
.typedef @array1  = array<@i32 10>
.typedef @array2  = array<@i32 4096>
```

Like structs, arrays may contain other structs, arrays or vectors, but not itself. It must have at least one element. Arrays of references are allowed.

`vector<T n>` is the **vector** type. It is designed for single-instruction multiple-data (SIMD) operations. Most modern desktop processors have SIMD capabilities. Vectors are used in very different ways compared to arrays. Vectors are usually small and are usually similar to the vector sizes supported by the machine.

Even today, architectures still do not agree upon any particular vector sizes. Mu only mandate the following three vector types to be implemented:

```
.typedef @4xi32     = vector<@i32 4>
.typedef @4xfloat   = vector<@float 4>
.typedef @2xdouble  = vector<@double 2>
```

The hybrid

`hybrid<F1 F2 ... V>` is a **hybrid** of a struct and an array. It starts with a *fixed part*: `F1`, `F2`, ... which is like a struct. It is followed by a *variable part*: an array of many elements of type `V`.

```
.typedef @hybrid1 = hybrid<@i64 @i32>
.typedef @hybrid2 = hybrid<@i8>
.typedef @hybrid3 = hybrid<@i64 @i64 @i64 @float>
```

In the above example, `@hybrid1` has one `@i64` field in its fixed part, and many `@i32` elements in its variable part. `@hybrid2` has an empty fixed part, and its variable parts are many `@i8` elements. `@hybrid3` has three `@i64` fields in its fixed part, and many `@float` elements in the variable part.

`hybrid` is *the only variable-size type* type in Mu whose size is determined at allocation site rather than the type itself. A hybrid must be allocated by special instructions, such as `NEWHYBRID`, which takes not only the type but also the length as its arguments.

```
%length1 = .....
%length2 = .....
%r1  = NEWHYBRID <@hybrid1 @i64> %length1   // @i64 is the length of %length1
%r2  = NEWHYBRID <@hybrid1 @i64> %length2   // @i64 is the length of %length2
```

In the above example, `%r1` and `%r2` refers to two different objects. Both have type `@hybrid1`, but the length of their variable parts are `%length1` and `%length2`, respectively.

Since the length cannot be determined by the type itself, it cannot be embedded in other aggregate types, not even other hybrids:

```
.typedef @some_struct = struct<@i64 @hybrid1 @hybrid2> // ERROR! cannot embed hybrids
```

Hybrid is the counterpart of the C99 structs with “flexible array elements”. In C99, you can write something like:

```
struct hybrid1 { int64_t f1; int32_t v[]; };
struct hybrid2 { int32_t v[]; };
struct hybrid3 { int64_t f1, f2, f3; float v[]; };

struct hybrid1 *p1 = malloc(sizeof(int64_t) + 1000*sizeof(int32_t));
struct hybrid1 *p2 = malloc(sizeof(int64_t) + 2000*sizeof(int32_t));
```

Once malloc-ed with enough memory, C can access the dynamically allocated “tail” elements.

The void type

`void` means “anything”, and can only be used as the target of references or pointers. For example:

```
.typedef @void = void
.typedef @refvoid   = ref<@void>
.typedef @irefvoid  = iref<@void>
.typedef @uptrvoid  = uptr<@void>
```

`ref<@void>` means the object reference can refer to any object. `iref<@void>` means the internal reference can refer to any memory location. `uptr<@void>` means it is... err... just a pointer, and has not been assigned a type yet.

Mu does not have the concept of "inheritance", but there are some "prefix rules" so that a reference may refer to some more complex objects than its `<T>` parameter. `void` is just the "simplest" type: no content at all.

You can allocate heap objects of the `void` type.

```
%r = NEW <@void>
```

Such objects have no contents, but each allocated `void` object is different, and compares equal (`EQ`) to only itself.

In Java, such use is like `Object o1 = new Object();`. There are some corner cases where such objects can be used as a "key" to identify something.

Other types

`stackref`, `threadref` and `framecursorref` refers to "special things" in Mu: stacks, threads and frame cursors. You will need the first two to start a Mu program, and need the third to perform stack introspection and on-stack replacement.

`weakref<T>` is the weak object reference type.

`tagref64` is the **tagged reference type**. It uses some clever bit-magic to reuse the NaN space of `double` to represent a tagged union of `double`, `int<52>` and a struct of `ref<void>` and `int<6>`.

`uptr<T>` and `ufuncptr<sig>` are **untraced (raw) pointers**. They are defined to be represented as integers of the pointer size, which is implementation-specific. For example, on a 64-bit implementation, it is 64 bits. But if you want to perform pointer arithmetic, you need to convert them to integers first.

You are unlikely to use raw pointers unless your program interacts with native programs (usually written in C). The garbage collector will not trace them: they are treated just like integers.

If you worked with x86 before, you may ask: Wait! Pointer is not just the address, but also its segment. Sorry, x86. But we see the trend is to move away from segmented architecture (x86_64 moved away from segments, too). For embedded systems that may have multiple address spaces, Mu is not designed for such systems, but supporting such architectures is an open topic.

1.5.4 Bonus section

Note: These contents should be moved to other chapters in the future. But if you are interested and patient enough, you can keep reading.

In fact, an internal reference refers to a "**memory location**" (discussed in later chapters) of type `T`. Memory location is a very important concept in Mu. It is a location in the Mu memory that can hold a Mu value. A field of an object is one kind of memory location. All memory accessing operations, such as `LOAD` and `STORE` directly work on internal references. This is different from JVM, where there are `getfield` and `setfield` instructions that work on object references.

If you worked with C before, it is the counterpart of the concept of "object". (What? You say C does not have "objects" but C++ does? Go ahead and read the C specification. In C, "object" means "a region of data storage" and does not mean object-oriented programming.) But the word "object" is used as a synonym of "heap object" in Mu. To avoid ambiguity, we use the word "memory location" instead.

1.6 Simple Functions

The constant and function syntax is defined in the Intermediate Representation chapter of the specification.

The Mu IR uses a variant of the static single assignment (SSA) or static single information (SSI) form. Mu IR has control flow graphs (CFG), each has many basic blocks, each then has many parameters and instructions. The main difference is that basic blocks take parameters, which are the counterpart of the PHI-nodes in SSA. At the end of a basic block, if it branches to another basic block, it must also specify the arguments to the destination, which are the counterpart of the SIGMA-nodes in SSI. There is no explicit PHI- or SIGMA-node. Instructions can only use global variables, the parameters of the basic block it is in, or the variables evaluated before that instruction in the same basic block. In other words, each basic block is a local scope and is like a single-exit "straight-line" function.

1.6.1 SSA Variable

In the Mu IR, as in SSA, a variable is defined in exactly one place and never redefined. For this reason, we still call them **SSA variables** since they are only assigned in one place.

In the Mu IR, variables are referred to by names, such as `@foo` or `%bar`.

Variables can be global or local. Global SSA variables are globally valid and never change. (Sorry but we still call them "variables".) Local SSA variables are only valid within a basic block and gets a value every time it is evaluated. Oh, did I say variables are defined in one *place*? Yes, they are, but this does not prevent them from being assigned multiple *times*. That is what "static" single assignment means.

1.6.2 Constant Definitions

Constant definitions, i.e. `.const`, are a kind of top-level definition. They construct values using literals.

```
.typedef @i8      = int<8>
.typedef @i16     = int<16>
.typedef @i32     = int<32>
.typedef @i64     = int<64>
.typedef @float   = float
.typedef @double  = double
.typedef @refi64 = ref<@i64>

.const @I8_10  <@i8>  = 10
.const @I16_10 <@i16> = 10
.const @I32_10 <@i32> = 10
.const @I64_10 <@i64> = 10
.const @MAGIC_NUMBER1 <@i64> = 0x123456789abcdef0
.const @MAGIC_NUMBER2 <@i64> = 0xfedcba9876543210
.const @MAGIC_NUMBER3 <@i64> = -0x8000000000000000

.const @F_PI  <@float>  = 3.14f
.const @D_2PI <@double> = 6.28d

.const @MY_CONSTANT_REF <@refi64> = NULL
```

On the left side of =, there are the name of the constant (such as `@I8_10`) and its type (such as `@i8`). On the right side it is, as you can guess, the **constant constructor**.

To construct an integer, you can write it in the decimal form or the hexadecimal form (add `0x` before). It may also have a sign. Since integers themselves in Mu do not have signs, the integer literal is just used to encode the bit pattern.

Mu uses the 2's complement representation for negative numbers, so `0xffffffff` and `-1` are the same if both are 32-bit.

To construct a floating point number, you can write it in the decimal form, with a decimal point (that is, 1.0, not 1), and append an `f` for float or a `d` for double. `nanf`, `+inff` and `-inff` will construct NaN, positive infinity and negative infinity of the `float` type. Replace the last `f` with `d` and it will be the `double` type.

```
.const @F_NAN  <@float> = nanf   // Mu will interpret it as an arbitrary NaN
.const @F_PINF <@float> = +inff  // positive infinity
.const @F_NINF <@float> = -inff  // negative infinity
```

If you are an FP number wizard, you can also explicitly specify the bit layout of an FP constant:

```
.const @D_1    <@double> = bitsd(0x3ff0000000000000)   // +1.0
.const @D_2    <@double> = bitsd(0x400c000000000000)   // +3.5
.const @D_PINF <@double> = bitsd(0x7ff0000000000000)   // +inf
.const @D_NAN  <@double> = bitsd(0x7ff0000000000001)   // nan (one possible encoding)
```

You can define constants of general reference types (`ref`, `iref`, `funcref`, `threadref`, `stackref` and `framecursorref`), too. But the only possible constant value is `NULL`.

```
.typedef @refi64  = ref<@i64>
.typedef @irefi64 = iref<@i64>
.funcsig @foo.sig = () -> ()
.typedef @foo.fr  = funcref<@foo.sig>
.typedef @tr      = threadref
.typedef @sr      = stackref
.typedef @fcr     = framecursorref

.const @NULLREF   <@refi64>  = NULL
.const @NULLIREF  <@irefi64> = NULL
.const @NULLFR    <@foo.fr>  = NULL
.const @NULLTR    <@tr>      = NULL
.const @NULLSR    <@sr>      = NULL
.const @NULLFCR   <@fcr>     = NULL
```

That is, you cannot define a constant reference to any heap object.

Note: Why there is no constant references to objects?

First of all, constants, as the name suggests, never change. If a constant refers to an object, the object is immortal! But the reason why we use the heap is to use GC, which eventually recycles the object.

Secondly, from the implementation's point of view, the advantage of using constants is that they can exist as immediate values in machine instructions, or be created by some machine code idioms (e.g. `xor rax, rax` makes rax 0, and the instruction decoder in modern processors (since IvyBridge) can eliminate such "idioms" in the front end), rather than being stored in the memory and loaded when needed (memory is slow nowadays compared to 20 years ago). But if the type is object reference, perhaps the only feasible way to implement such constant is to store it in the memory so that copying GC can update it when the referenced object is moved. Non-copying GC sucks, because the VM will eventually die of heap fragmentation. (R.I.P. lighttpd. You know, C programmers are responsible for memory management. If C's malloc cannot manage the memory well and kills long-running servers, we should use a VM with copying GC, instead. If usual VMs perform too bad, that's why we build the Mu micro VM.) But if the GC ends up modifying the machine code to fix the reference, it will be too painful.

If we really need some permanent global memory space, Mu has another top-level definition: global cells, i.e. `.global` (it will be discussed in details when we talk about memory access). Global cells are memory locations: they are mutable. They can be loaded and stored, and they are permanent. Just store an object reference in a global cell and it has all the benefits of constant references.

For other references, constant function reference is unnecessary because the name of the function is already a constant function reference. Stacks are similar to heap objects. Threads and frame cursors have their own lifecycles, so you can't possibly create such constants that remain valid.

However, pointers are not references. They are just integers and can be constructed as integers.

```
.typedef @i64    = int<64>
.typedef @ptri64 = uptr<@i64>

.const @MY_POINTER <@ptri64> = 0x123456789000

.funcsig @bar.sig = () -> ()
.typedef @bar.fp  = ufuncptr<@bar.sig>

// The address can be looked up by dlsym.
.const @MY_FUNCTION_POINTER <@bar.fp> = 0x7fff00001230
```

Mu support constants of non-hybrid composite types, too. A composite constant is constructed by referring to other constants. Please put as many fields/elements as there should be.

```
.typedef @i32     = int<32>
.typedef @struct1 = struct<@i32 @i32 @i32>
.typedef @array1  = array<@i32 2>
.typedef @vector1 = vector<@i32 4>

.const @I32_1 <@i32> = 1
.const @I32_2 <@i32> = 2
.const @I32_3 <@i32> = 3
.const @I32_4 <@i32> = 4
.const @S1    <@struct1> = { @I32_1 @I32_2 @I32_3 }
.const @A1    <@array1>  = { @I32_1 @I32_2 }
.const @V1    <@vector1> = { @I32_1 @I32_2 @I32_3 @I32_4 }

.typedef @struct2 = struct<@struct1 @i64>

.const @S2    <@struct2> = { @S1 @I32_4 }   // correct

.const @WRONG <@struct2> = { {@I32_1 @I32_2 @I32_3} @I32_4 }   // ERROR: cannot nest␣
↪braces. Define separately
```

But it is *not recommended to use constants of composite types, unless they are small*. Mu may not be able to allocate big values into registers, in which case it may perform stupid copying. The micro VM may not be smart enough to do too much optimisation.

1.6.3 Function definition

TODO