CS4120/4121/5120/5121—Spring 2023

Eta High-Level ABI Specification
Cornell University
Version of March 9, 2023

0 Changes

- None yet; watch this space.

1 Introduction

The programs compilers produce are rarely standalone—they have to interface with the operating system's libraries for things like I/O, memory management, and GUIs. In order to do this, programs must adhere to certain conventions. These conventions are usually specified by platform vendors (e.g., Microsoft, Apple, Intel, and The Linux Foundation) as part of what's called the platform's *application binary interface* (ABI).

These specifications are usually extremely helpful to compiler writers, because they remove the pain of having to resolve ambiguities (a process that you may already have found difficult!).

In PA5, your compiler will interface with the Eta standard runtime that we will provide, but some of the details of that connection already need to be reflected in your PA4 IR. This document specifies the ABI your Eta compilers should follow to properly interface with the library. It assumes a 64-bit system and is meant to be similar to how C function calls work on x86-64 on Windows, Linux and Mac OS X in 64-bit code.

2 Mangling function names

To better support separate compilation, your implementation must emit symbol names for functions and procedures that include type information exactly as specified below. This allows the linker to catch type errors when modules disagree about the types of functions.

2.1 Function and procedure names

The encoding of a procedure or function name is the sequence of the following:

- the string '`_I`'
- the name of the function, encoded as described below
- the underscore character, '`_`'
- the encoding of the return type, or '`p`' if this is a procedure, and
- encodings of types of each of the arguments, if there are any.

To encode function names all you need to do is replace a single underscore character with two of them (and 2 will get replaced with 4, and so on). Thus, a single underscore character can be known to separate out the function name from the argument information.

Type names are encoded as following:

- `int` is encoded as '`i`'
- `bool` is encoded as '`b`'
- An array of type τ is encoded as '`a`' followed by the encoding of the element type.
- A tuple, from a function returning multiple results, is encoded as '`t`' followed by the number of arguments returned followed by the encodings of the types returned.

2.2 Examples

Declaration	Symbol name
`main(args: int[][])`	`_Imain_paai`
`unparseInt(n: int): int[]`	`_IunparseInt_aii`
`parseInt(str: int[]): int, bool`	`_IparseInt_t2ibai`
`eof(): bool`	`_Ieof_b`
`gcd(a: int, b: int): int`	`_Igcd_iii`
`multiple__underScores()`	`_Imultiple____underScores_p`

2.3 Special names

You will need to use the runtime library to allocate heap memory for arrays. To do this, you need to call (in your IR) the function `_eta_alloc`, passing it a single integer giving the number of bytes to allocate. This function will return the memory address corresponding to the first byte of the allocated memory.

If you detect an array index out of bounds access, you can call the `_eta_out_of_bounds` function, which will print an error message and abort execution.

Note that these names will not conflict with a source-level Eta function or procedure declaration, as they do not follow the mangling convention.

Nothing else requires special handling; for example, your main function will be expected to be called `_Imain_paai` as it is by above rules.

3 Memory layout of arrays

All types have sizes that are multiples of 8 and native alignment of 8. Arrays are therefore laid out sequentially, with no need for padding.

To implement the `length` operation on arrays, their size is stored at index "-1" in the array: that is, immediately before cell 0. An array value is a reference to the memory location of cell 0.

Note: the blocks returned by `_eta_alloc` will always be at least 8-aligned.

4 Passing arguments and returning results

At present, two notable conventions for passing and returning function arguments exist: the independently developed Microsoft calling conventions, and the System V ABI, used by Mac OS, Linux, and other Unix-based operating systems.

In this course, we require that you support the System V ABI so that the course staff can run your submissions on the released Linux virtual machine.

Fortunately, given the right compiler architecture, it is relatively straightforward to support multiple calling conventions, which may desirable if the members of your group use different operating systems. This section therefore presents an overview of the System V ABI and important platform-dependent differences that you will need to consider if you choose to support operating systems other than Linux. Since this is only a high-level summary, you should also look at the actual specifications, linked at the end of this document.

4.1 Registers

In the System V ABI, six registers are designated for passing parameters. In order, these are: `rdi`, `rsi`, `rdx`, `rcx`, `r8`, and `r9`. The register `rax` is used to hold the return value of a function that returns a single result. For a function that returns two or more results, register `rax` contains the first result and register `rdx` contains the second result. The registers `rbx`, `rbp`, and `r12`–`r15` are *callee-saved*, which means that a function that uses these registers must restore their value before returning. All registers not classified as callee-saved are *caller-saved*, which means that their contents are potentially destroyed when calling a function (and their contents must therefore be saved if they are still needed after the function call).

On Windows, four registers are used to pass parameters: `rcx`, `rdx`, `r8`, and `r9`. Again, the register `rax` holds the return value of a function that returns a single result. There is no register assigned by Windows to the second result, but `rdx` is also available on Windows for this purpose. Registers `rdi`, `rsi`, `rbx`, `rbp`, and `r12`–`r15` are callee-saved.

4.2 Stack

When the number of function arguments exceeds six (or four on Windows), the remaining arguments are pushed to the stack right-to-left so that the last argument is the farthest away from the top of the stack.

All stack entries are aligned to 8-byte boundaries, since the stack pointer `rsp` is required by the ISA to be aligned to the size of items pushed and popped. The virtual machine may be more forgiving about stack misalignment than what is required by the ISA, but you should follow the ISA requirements. Care must also be taken to ensure that the stack is aligned to a 16 byte boundary before issuing a function call. Your application may crash if it calls a library function without an aligned stack.

On Windows, any function must furthermore allocate a temporary stack region named *shadow space* before issuing a call to another function. Conceptually, this is the stack space that would have been taken up by the first four parameters, had they not been passed using registers. However, this shadow region is always 32 bytes large (even if the function uses fewer than four parameters).

4.3 Passing and returning Eta values

Generally, `int` and `bool` values are passed directly using registers. Arrays have reference semantics, hence they are passed as 64-bit pointers using the same architectural registers. There are two cases when values are not passed using registers: the first one is the aforementioned situation when a function takes too many arguments.

The second case occurs when calling functions that return three or more results. Here, the caller must allocate space for the third and following return values, ordinarily within its own stack frame. It passes a pointer to that memory region as an extra argument to the callee, before all the ordinary Eta-language arguments. The callee should return the first and second results in `rax` and `rdx`, but save the third and following values to the specified location. For a function returning n values, the memory region must be at least $8 \times (n-2)$ bytes long to be able to store all the values.

We recommend that your register allocator computes a static stack frame layout for each function, accounting for all the memory needed to store caller-save registers, tuple return values, function arguments, as well as unused memory needed to ensure 16-byte alignment and (if Windows is supported) the shadow space on Windows targets. Having a static layout means that the stack pointer is not manually modified in the body of a function, and this makes it easier to ensure that the stack pointer is 16-byte aligned.

One such layout is the following (with memory addresses decreasing downwards).

Return address
Optional: saved frame pointer
Callee-saved registers
Spilled `TEMP`s
Optional: 8 bytes for alignment
Scratch space for functions that return tuples
Scratch space for arguments delivered on the stack
Optional: 32-bytes of shadow space

Some of these regions are not always necessary. For instance, a *leaf* function (i.e., a function that never calls other functions) does not need to allocate alignment or shadow space. If your compiler does not use an explicit frame pointer, there is no need to store its value directly after the return address. In this case, note that the `rbp` register is callee-saved, hence its value must be restored by any function that changes its value.

The runtime library comes with few basic Eta example programs and sample assembly output for all three platforms, which may be useful as a reference. The sample assembly code uses the stack layout shown here.

5 See also

You may find it useful to look at the specifications used in real-life systems, See AMD64 Application Binary Interface (v 0.99). The following document may also be useful: x86-64 Machine-Level Programming. Groups wishing to support Microsoft calling conventions (this is optional) may want to look at the MSDN documentation.