Iota$_9$ High-Level ABI Specification

Computer Science 4120
Cornell University

Version of November 1, 2009

Changes

- October 24: Updated for PA5 with boolean type handling simplications, `_I_outOfBounds_p()`, and caller/callee save conventions.

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, or The Linux Foundation) as part of what's often 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 on the spot (a process that you have likely found already to be very difficult).

In PA5, Your compiler will interface with the Iota$_9$standard runtime that we will provide; but some of the details of that connection will need to be reflected in your PA4 IR. This document specifies the ABI your Iota$_9$compilers should follow to properly interface with the library. It assumes a 32-bit system, as the common denominator, and is meant to be similar to how C function calls work on IA-32 on Windows, Linux and OS X.

Mangling function names

To better support separate compilation, your implementation should emit symbol names for functions and procedures that include type information exactly as specified below. This is so that the linker can issue appropriate errors in the event of disagreement between modules.

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
- 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 is encoded as `t` followed by the tuple length, and then the encodings of each component type

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`
`an_example(a:((int,int),(bool,bool,bool))):int[]`	`_Ian__example_ait2t2iit3bbb`
`multiple__underScores()`	`_Imultiple____underScores_p`

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 `_I_alloc_i`, 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 `_I_outOfBounds_p` function, which will print an error message and abort execution.

Note that these names will not conflict with a source-level Iota$_9$function or procedure declaration, as identifiers are not permitted to start with `_`.

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

Memory layout of arrays and tuples

While in tuples, the type `bool` should be represented as a 4-byte value, with the value 0 representing false, and value 1 representing true. Because of this, all types have sizes that are multiples of 4 and native alignment of 4. Hence, arrays and tuples are laid out sequentially, with no need for padding.

To implement the `length` operation on arrays, their size should be stored in the "-1"st index of the array, e.g. immediately before the payload, with array references pointing to the cell 0.

Note: the blocks returned by `_I_alloc_i` will always be at least 4-aligned.

Examples

Type	Offsets of components	Alignment
`(bool, bool)`	0, 4	1
`(int, bool)`	0, 4	4
`(bool, bool, int)`	0, 4, 8	4
`((bool,bool), bool, int, bool, bool, int[])`	0, 4, 8, 12, 16, 20	4

Passing in arguments and returning results

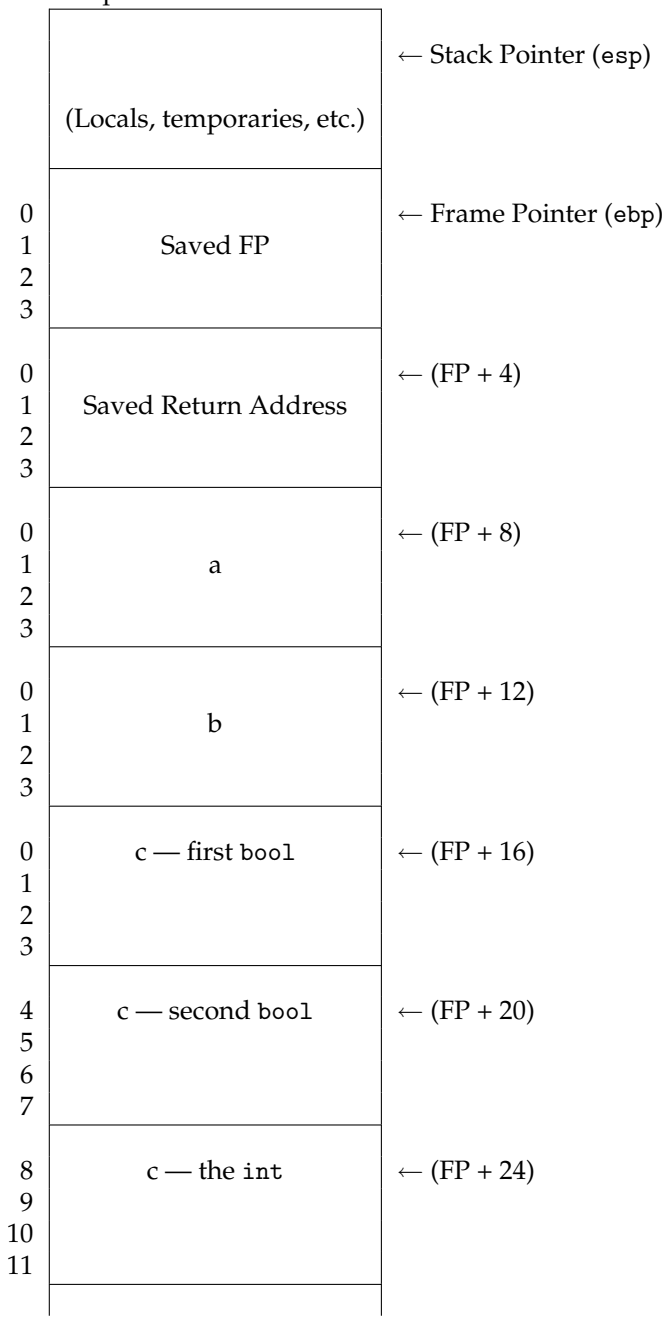
All arguments are pushed to the stack right-to-left (so the first argument is closer to the top of the stack), and all are aligned to 4-byte boundaries. Tuples are passed by value (i.e., they are fully copied). Since arrays have reference semantics, you just pass the pointer to the same array.

`int` and `bool` return values should be returned in the `eax` architectural register. The same applies to arrays.

For functions returning tuple types, the caller must allocate space for them on its own accord, and pass a pointer to that area as an extra argument, before all the actual arguments (so closest to the top of the stack), and the callee should copy the returned tuple into that address.

You are encouraged to manage the frame pointer the same way C compilers do, as in the example below; this is not necessary for interoperability, but will make tools like `gdb` more useful.

Shown below is how a stack frame of a function receiving arguments `(a:int, b:bool, c:(bool, bool, int)` may look like. Note that the stack grows towards lower memory addresses; and this picture has lower addresses on top.



Register ownership

The registers `eax`,`edx`, and `ecx` are caller save. Registers `ebx`, `esi`, `edi` and `ebp` are callee save.

See also

You may find it useful to look at the specifications used in real-life systems, though this specification is meant to be sufficient standalone:

System V Application Binary Interface - Intel386 Architecture Processor Supplement, Fourth Edition (section 3, in particular 3-9)

Mac OS X ABI Function Call Guide — IA-32 Function Calling Conventions (based on the above; note that the way small structures are returned is different from what is specified here for small tuples)

C++ Language Reference: Calling Conventions (from Microsoft Visual Studio 2010 manual) (_cdecl is the most similar to conventions we used; but other modes may be of interest as they cover design alternatives. Unfortunately this document is not very detailed).