Working with LLVM

This is not a replacement for the LLVM documentation, but a collection of tips for working on LLVM for Julia.

Overview of Julia to LLVM Interface

Julia dynamically links against LLVM by default. Build with `USE_LLVM_SHLIB=0` to link statically.

The code for lowering Julia AST to LLVM IR or interpreting it directly is in directory `src/`.

File	Description
`builtins.c`	Builtin functions
`ccall.cpp`	Lowering `ccall`
`cgutils.cpp`	Lowering utilities, notably for array and tuple accesses
`codegen.cpp`	Top-level of code generation, pass list, lowering builtins
`debuginfo.cpp`	Tracks debug information for JIT code
`disasm.cpp`	Handles native object file and JIT code diassembly
`gf.c`	Generic functions
`intrinsics.cpp`	Lowering intrinsics
`llvm-simdloop.cpp`	Custom LLVM pass for `@simd`
`sys.c`	I/O and operating system utility functions

Some of the `.cpp` files form a group that compile to a single object.

The difference between an intrinsic and a builtin is that a builtin is a first class function that can be used like any other Julia function. An intrinsic can operate only on unboxed data, and therefore its arguments must be statically typed.

Alias Analysis

Julia currently uses LLVM's Type Based Alias Analysis. To find the comments that document the inclusion relationships, look for `static MDNode*` in `src/codegen.cpp`.

The `-O` option enables LLVM's Basic Alias Analysis.

Building Julia with a different version of LLVM

The default version of LLVM is specified in `deps/Versions.make`. You can override it by creating a file called `Make.user` in the top-level directory and adding a line to it such as:

```
LLVM_VER = 6.0.1
```

Besides the LLVM release numerals, you can also use `LLVM_VER = svn` to build against the latest development version of LLVM.

You can also specify to build a debug version of LLVM, by setting either `LLVM_DEBUG = 1` or `LLVM_DEBUG = Release` in your `Make.user` file. The former will be a fully unoptimized build of LLVM and the latter will produce an optimized build of LLVM. Depending on your needs the latter will suffice and it quite a bit faster. If you use `LLVM_DEBUG = Release` you will also want to set `LLVM_ASSERTIONS = 1` to enable diagonstics for different passes. Only `LLVM_DEBUG = 1` implies that option by default.

Passing options to LLVM

You can pass options to LLVM via the environment variable `JULIA_LLVM_ARGS`. Here are example settings using `bash` syntax:

- `export JULIA_LLVM_ARGS = -print-after-all` dumps IR after each pass.
- `export JULIA_LLVM_ARGS = -debug-only=loop-vectorize` dumps LLVM `DEBUG(...)` diagnostics for loop vectorizer. If you get warnings about "Unknown command line argument", rebuild LLVM with `LLVM_ASSERTIONS = 1`.

Debugging LLVM transformations in isolation

On occasion, it can be useful to debug LLVM's transformations in isolation from the rest of the Julia system, e.g. because reproducing the issue inside `julia` would take too long, or because one wants to take advantage of LLVM's tooling (e.g. bugpoint). To get unoptimized IR for the entire system image, pass the `--output-unopt-bc unopt.bc` option to the system image build process, which will output the

unoptimized IR to an `unopt.bc` file. This file can then be passed to LLVM tools as usual. `libjulia` can function as an LLVM pass plugin and can be loaded into LLVM tools, to make julia-specific passes available in this environment. In addition, it exposes the `-julia` meta-pass, which runs the entire Julia pass-pipeline over the IR. As an example, to generate a system image, one could do:

```
opt -load libjulia.so -julia -o opt.bc unopt.bc
llc -o sys.o opt.bc
cc -shared -o sys.so sys.o
```

This system image can then be loaded by `julia` as usual.

Alternatively, you can use `--output-jit-bc jit.bc` to obtain a trace of all IR passed to the JIT. This is useful for code that cannot be run as part of the sysimg generation process (e.g. because it creates unserializable state). However, the resulting `jit.bc` does not include sysimage data, and can thus not be used as such.

It is also possible to dump an LLVM IR module for just one Julia function, using:

```
fun, T = +, Tuple{Int,Int} # Substitute your function of interest here
optimize = false
open("plus.ll", "w") do file
    println(file, InteractiveUtils._dump_function(fun, T, false, false, false, true,
end
```

These files can be processed the same way as the unoptimized sysimg IR shown above.

Improving LLVM optimizations for Julia

Improving LLVM code generation usually involves either changing Julia lowering to be more friendly to LLVM's passes, or improving a pass.

If you are planning to improve a pass, be sure to read the LLVM developer policy. The best strategy is to create a code example in a form where you can use LLVM's `opt` tool to study it and the pass of interest in isolation.

1. Create an example Julia code of interest.
2. Use `JULIA_LLVM_ARGS = -print-after-all` to dump the IR.
3. Pick out the IR at the point just before the pass of interest runs.
4. Strip the debug metadata and fix up the TBAA metadata by hand.

The last step is labor intensive. Suggestions on a better way would be appreciated.

The jlcall calling convention

Julia has a generic calling convention for unoptimized code, which looks somewhat as follows:

```
jl_value_t *any_unoptimized_call(jl_value_t *, jl_value_t **, int);
```

where the first argument is the boxed function object, the second argument is an on-stack array of arguments and the third is the number of arguments. Now, we could perform a straightforward lowering and emit an alloca for the argument array. However, this would betray the SSA nature of the uses at the call site, making optimizations (including GC root placement), significantly harder. Instead, we emit it as follows:

```
%bitcast = bitcast @any_unoptimized_call to %jl_value_t *(*)(%jl_value_t *, %jl_value
call cc 37 %jl_value_t *%bitcast(%jl_value_t *%arg1, %jl_value_t *%arg2)
```

The special `cc 37` annotation marks the fact that this call site is really using the jlcall calling convention. This allows us to retain the SSA-ness of the uses throughout the optimizer. GC root placement will later lower this call to the original C ABI. In the code the calling convention number is represented by the `JLCALL_F_CC` constant. In addition, there is the `JLCALL_CC` calling convention which functions similarly, but omits the first argument.

GC root placement

GC root placement is done by an LLVM pass late in the pass pipeline. Doing GC root placement this late enables LLVM to make more aggressive optimizations around code that requires GC roots, as well as allowing us to reduce the number of required GC roots and GC root store operations (since LLVM doesn't understand our GC, it wouldn't otherwise know what it is and is not allowed to do with values stored to the GC frame, so it'll conservatively do very little). As an example, consider an error path

```
if some_condition()
    #= Use some variables maybe =#
    error("An error occurred")
end
```

During constant folding, LLVM may discover that the condition is always false, and can remove the basic block. However, if GC root lowering is done early, the GC root slots used in the deleted block, as well as any values kept alive in those slots only because they were used in the error path, would be kept alive by LLVM. By doing GC root lowering late, we give LLVM the license to do any of its usual optimizations (constant folding, dead code elimination, etc.), without having to worry (too much) about which values

may or may not be GC tracked.

However, in order to be able to do late GC root placement, we need to be able to identify a) which pointers are GC tracked and b) all uses of such pointers. The goal of the GC placement pass is thus simple:

Minimize the number of needed GC roots/stores to them subject to the constraint that at every safepoint, any live GC-tracked pointer (i.e. for which there is a path after this point that contains a use of this pointer) is in some GC slot.

Representation

The primary difficulty is thus choosing an IR representation that allows us to identify GC-tracked pointers and their uses, even after the program has been run through the optimizer. Our design makes use of three LLVM features to achieve this:

- Custom address spaces
- Operand Bundles
- Non-integral pointers

Custom address spaces allow us to tag every point with an integer that needs to be preserved through optimizations. The compiler may not insert casts between address spaces that did not exist in the original program and it must never change the address space of a pointer on a load/store/etc operation. This allows us to annotate which pointers are GC-tracked in an optimizer-resistant way. Note that metadata would not be able to achieve the same purpose. Metadata is supposed to always be discardable without altering the semantics of the program. However, failing to identify a GC-tracked pointer alters the resulting program behavior dramatically - it'll probably crash or return wrong results. We currently use three different address spaces (their numbers are defined in `src/codegen_shared.cpp`):

- GC Tracked Pointers (currently 10): These are pointers to boxed values that may be put into a GC frame. It is loosely equivalent to a `jl_value_t*` pointer on the C side. N.B. It is illegal to ever have a pointer in this address space that may not be stored to a GC slot.
- Derived Pointers (currently 11): These are pointers that are derived from some GC tracked pointer. Uses of these pointers generate uses of the original pointer. However, they need not themselves be known to the GC. The GC root placement pass MUST always find the GC tracked pointer from which this pointer is derived and use that as the pointer to root.
- Callee Rooted Pointers (currently 12): This is a utility address space to express the notion of a callee rooted value. All values of this address space MUST be storable to a GC root (though it is possible to relax this condition in the future), but unlike the other pointers need not be rooted if

passed to a call (they do still need to be rooted if they are live across another safepoint between the definition and the call).
- Pointers loaded from tracked object (currently 13): This is used by arrays, which themselves contain a pointer to the managed data. This data area is owned by the array, but is not a GC-tracked object by itself. The compiler guarantees that as long as this pointer is live, the object that this pointer was loaded from will keep being live.

Invariants

The GC root placement pass makes use of several invariants, which need to be observed by the frontend and are preserved by the optimizer.

First, only the following address space casts are allowed:

- 0->{Tracked,Derived,CalleeRooted}: It is allowable to decay an untracked pointer to any of the others. However, do note that the optimizer has broad license to not root such a value. It is never safe to have a value in address space 0 in any part of the program if it is (or is derived from) a value that requires a GC root.
- Tracked->Derived: This is the standard decay route for interior values. The placement pass will look for these to identify the base pointer for any use.
- Tracked->CalleeRooted: Addrspace CalleeRooted serves merely as a hint that a GC root is not required. However, do note that the Derived->CalleeRooted decay is prohibited, since pointers should generally be storable to a GC slot, even in this address space.

Now let us consider what constitutes a use:

- Loads whose loaded values is in one of the address spaces
- Stores of a value in one of the address spaces to a location
- Stores to a pointer in one of the address spaces
- Calls for which a value in one of the address spaces is an operand
- Calls in jlcall ABI, for which the argument array contains a value
- Return instructions.

We explicitly allow load/stores and simple calls in address spaces Tracked/Derived. Elements of jlcall argument arrays must always be in address space Tracked (it is required by the ABI that they are valid `jl_value_t*` pointers). The same is true for return instructions (though note that struct return arguments are allowed to have any of the address spaces). The only allowable use of an address space CalleeRooted pointer is to pass it to a call (which must have an appropriately typed operand).

Further, we disallow `getelementptr` in addrspace Tracked. This is because unless the operation is a

noop, the resulting pointer will not be validly storable to a GC slot and may thus not be in this address space. If such a pointer is required, it should be decayed to addrspace Derived first.

Lastly, we disallow `inttoptr/ptrtoint` instructions in these address spaces. Having these instructions would mean that some `i64` values are really GC tracked. This is problematic, because it breaks that stated requirement that we're able to identify GC-relevant pointers. This invariant is accomplished using the LLVM "non-integral pointers" feature, which is new in LLVM 5.0. It prohibits the optimizer from making optimizations that would introduce these operations. Note we can still insert static constants at JIT time by using `inttoptr` in address space 0 and then decaying to the appropriate address space afterwards.

Supporting `ccall`

One important aspect missing from the discussion so far is the handling of `ccall`. `ccall` has the peculiar feature that the location and scope of a use do not coincide. As an example consider:

```
A = randn(1024)
ccall(:foo, Cvoid, (Ptr{Float64},), A)
```

In lowering, the compiler will insert a conversion from the array to the pointer which drops the reference to the array value. However, we of course need to make sure that the array does stay alive while we're doing the `ccall`. To understand how this is done, first recall the lowering of the above code:

```
return $(Expr(:foreigncall, :(:foo), Cvoid, svec(Ptr{Float64}), 0, :(:ccall), Expr(
```

The last `:(A)`, is an extra argument list inserted during lowering that informs the code generator which Julia level values need to be kept alive for the duration of this `ccall`. We then take this information and represent it in an "operand bundle" at the IR level. An operand bundle is essentially a fake use that is attached to the call site. At the IR level, this looks like so:

```
call void inttoptr (i64 ... to void (double*)*)(double* %5) [ "jl_roots"(%jl_value_t
```

The GC root placement pass will treat the `jl_roots` operand bundle as if it were a regular operand. However, as a final step, after the GC roots are inserted, it will drop the operand bundle to avoid confusing instruction selection.

Supporting `pointer_from_objref`

`pointer_from_objref` is special because it requires the user to take explicit control of GC rooting. By

our above invariants, this function is illegal, because it performs an address space cast from 10 to 0. However, it can be useful, in certain situations, so we provide a special intrinsic:

```
declared %jl_value_t *julia.pointer_from_objref(%jl_value_t addrspace(10)*)
```

which is lowered to the corresponding address space cast after GC root lowering. Do note however that by using this intrinsic, the caller assumes all responsibility for making sure that the value in question is rooted. Further this intrinsic is not considered a use, so the GC root placement pass will not provide a GC root for the function. As a result, the external rooting must be arranged while the value is still tracked by the system. I.e. it is not valid to attempt to use the result of this operation to establish a global root - the optimizer may have already dropped the value.

Keeping values alive in the absence of uses

In certain cases it is necessary to keep an object alive, even though there is no compiler-visible use of said object. This may be case for low level code that operates on the memory-representation of an object directly or code that needs to interface with C code. In order to allow this, we provide the following intrinsics at the LLVM level:

```
token @llvm.julia.gc_preserve_begin(...)
void @llvm.julia.gc_preserve_end(token)
```

(The `llvm.` in the name is required in order to be able to use the `token` type). The semantics of these intrinsics are as follows: At any safepoint that is dominated by a `gc_preserve_begin` call, but that is not not dominated by a corresponding `gc_preserve_end` call (i.e. a call whose argument is the token returned by a `gc_preserve_begin` call), the values passed as arguments to that `gc_preserve_begin` will be kept live. Note that the `gc_preserve_begin` still counts as a regular use of those values, so the standard lifetime semantics will ensure that the values will be kept alive before entering the preserve region.

« System Image Building printf() and stdio in the Julia runtime »