Static analyzer annotations for GC correctness in C code

Running the analysis

The analyzer plugin that drives the anlysis ships with julia. Its source code can be found in `src/clangsa`. Running it requires the clang dependency to be build. Set the `BUILD_LLVM_CLANG` variable in your Make.user in order to build an appropriate version of clang. You may also want to use the prebuilt binaries using the `USE_BINARYBUILDER_LLVM` options. Afterwards, running the analysis over the source tree is as simple as running `make -C src analyzegc`.

General Overview

Since Julia's GC is precise, it needs to maintain correct rooting information for any value that may be referenced at any time GC may occur. These places are known as `safepoints` and in the function local context, we extend this designation to any function call that may recursively end up at a safepoint.

In generated code, this is taken care of automatically by the GC root placement pass (see the chapter on GC rooting in the LLVM codegen devdocs). However, in C code, we need to inform the runtime of any GC roots manually. This is done using the following macros:

```
// The value assigned to any slot passed as an argument to these
// is rooted for the duration of this GC frame.
JL_GC_PUSH{1,...,6}(args...)
// The values assigned into the size `n` array `rts` are rooted
// for the duration of this GC frame.
JL_GC_PUSHARGS(rts, n)
// Pop a GC frame
JL_GC_POP
```

If these macros are not used where they need to be, or they are used incorrectly, the result is silent memory corruption. As such it is very important that they are placed correctly in all applicable code.

As such, we employ static analysis (and in particular the clang static analyzer) to help ensure that these macros are used correctly. The remainder of this document gives an overview of this static analysis and describes the support needed in the julia code base to make things work.

GC Invariants

There is two simple invariants correctness:

- All `GC_PUSH` calls need to be followed by an appropriate `GC_POP` (in practice we enforce this at the function level)
- If a value was previously not rooted at any safepoint, it may no longer be referenced afterwards

Of course the devil is in the details here. In particular to satisfy the second of the above conditions, we need to know:

- Which calls are safepoints and which are not
- Which values are rooted at any given safepoint and which are not
- When is a value referenced

For the second point in particular, we need to know which memory locations will be considered rooting at runtime (i.e. values assigned to such locations are rooted). This includes locations explicitly designated as such by passing them to one of the `GC_PUSH` macros, globally rooted locations and values, as well as any location recursively reachable from one of those locations.

Static Analysis Algorithm

The idea itself is very simple, although the implementation is quite a bit more complicated (mainly due to a large number of special cases and intricacies of C and C++). In essence, we keep track of all locations that are rooting, all values that are rootable and any expression (assignments, allocations, etc) affect the rootedness of any rootable values. Then, at any safepoint, we perform a "symbolic GC" and poison any values that are not rooted at said location. If these values are later referenced, we emit an error.

The clang static analyzer works by constructing a graph of states and exploring this graph for sources of errors. Several nodes in this graph are generated by the analyzer itself (e.g. for control flow), but the definitions above augment this graph with our own state.

The static analyzer is interprocedural and can analyze control flow across function boundaries. However, the static analyzer is not fully recursive and makes heuristic decisions about which calls to explore (additionally some calls are cross-translation unit and invisible to the analyzer). In our case, our definition of correctness requires total information. As such, we need to annotate the prototypes of all function calls with whatever information the analysis required, even if that information would otherwise be available by interprocedural static analysis.

Luckily however, we can still use this interprocedural analysis to ensure that the annotations we place on a given function are indeed correct given the implementation of said function.

The analyzer annotations

These annotations are found in src/support/analyzer_annotations.h. The are only active when the analyzer is being used and expand either to nothing (for prototype annotations) or to no-ops (for function like annotations).

JL_NOTSAFEPOINT

This is perhaps the most common annotation, and should be placed on any function that is known not to possibly lead to reaching a GC safepoint. In general, it is only safe for such a function to perform arithmetic, memory accesses and calls to functions either annotated `JL_NOTSAFEPOINT` or otherwise known not to be safepoints (e.g. function in the C standard library, which are hardcoded as such in the analyzer)

It is valid to keep values unrooted across calls to any function annotated with this attribute:

Usage Example:

```
void jl_get_one() JL_NOTSAFEPOINT {
  return 1;
}

jl_value_t *example() {
  jl_value_t *val = jl_alloc_whatever();
  // This is valid, even though `val` is unrooted, because
  // jl_get_one is not a safepoint
  jl_get_one();
  return val;
}
```

JL_MAYBE_UNROOTED/JL_ROOTS_TEMPORARILY

When `JL_MAYBE_UNROOTED` is annotated as an argument on a function, indicates that said argument may be passed, even if it is not rooted. In the ordinary course of events, the julia ABI guarantees that callers root values before passing them to callees. However, some functions do not follow this ABI and allow values to be passed to them even though they are not rooted. Note however, that this does not automatically imply that said argument will be preserved. The `ROOTS_TEMPORARILY` annotation provides the stronger guarantee that, not only may the value be unrooted when passed, it will also be preserved across any internal safepoints by the callee.

Note that `JL_NOTSAFEPOINT` essentially implies `JL_MAYBE_UNROOTED/JL_ROOTS_TEMPORARILY`, because

the rootedness of an argument is irrelevant if the function contains no safepoints.

One additional point to note is that these annotations apply on both the caller and the callee side. On the caller side, they lift rootedness restrictions that are normally required for julia ABI functions. On the callee side, they have the reverse effect of preventing these arguments from being considered implicitly rooted.

If either of these annotations is applied to the function as a whole, it applies to all arguments of the function. This should generally only be necessary for varargs functions.

Usage example:

```
JL_DLLEXPORT void JL_NORETURN jl_throw(jl_value_t *e JL_MAYBE_UNROOTED);
jl_value_t *jl_alloc_error();

void example() {
  // The return value of the allocation is unrooted. This would normally
  // be an error, but is allowed because of the above annotation.
  jl_throw(jl_alloc_error());
}
```

JL_PROPAGATES_ROOT

This annotation is commonly found on accessor functions that return one rootable object stored within another. When annotated on a function argument, it tells the analyzer that the root for that argument also applies to the value returned by the function.

Usage Example:

```
jl_value_t *jl_svecref(jl_svec_t *t JL_PROPAGATES_ROOT, size_t i) JL_NOTSAFEPOINT;

size_t example(jl_svec_t *svec) {
  jl_value_t *val = jl_svecref(svec, 1)
  // This is valid, because, as annotated by the PROPAGATES_ROOT annotation,
  // jl_svecref propagates the rooted-ness from `svec` to `val`
  jl_gc_safepoint();
  return jl_unbox_long(val);
}
```

JL_ROOTING_ARGUMENT/JL_ROOTED_ARGUMENT

This is essentially the assignment counterpart to `JL_PROPAGATES_ROOT`. When assigning a value to a field of another value that is already rooted, the assigned value will inherit the root of the value it is assigned into.

Usage Example:

```
void jl_svecset(void *t JL_ROOTING_ARGUMENT, size_t i, void *x JL_ROOTED_ARGUMENT)


size_t example(jl_svec_t *svec) {
  jl_value_t *val = jl_box_long(10000);
  jl_svecset(svec, val);
  // This is valid, because the annotations imply that the
  // jl_svecset propagates the rooted-ness from `svec` to `val`
  jl_gc_safepoint();
  return jl_unbox_long(val);
}
```

JL_GC_DISABLED

This annotation implies that this function is only called with the GC runtime-disabled. Functions of this kind are most often encountered during startup and in the GC code itself. Note that this annotation is checked against the runtime enable/disable calls, so clang will know if you lie. This is not a good way to disable processing of a given function if the GC is not actually disabled (use `ifdef __clang_analyzer__` for that if you must).

Usage example:

```
void jl_do_magic() JL_GC_DISABLED {
  // Wildly allocate here with no regard for roots
}

void example() {
  int en = jl_gc_enable(0);
  jl_do_magic();
  jl_gc_enable(en);
}
```

JL_REQUIRE_ROOTED_SLOT

This annotation requires the caller to pass in a slot that is rooted (i.e. values assigned to this slot will be

rooted).

Usage example:

```
void jl_do_processing(jl_value_t **slot JL_REQUIRE_ROOTED_SLOT) {
  *slot = jl_box_long(1);
  // Ok, only, because the slot was annotated as rooting
  jl_gc_safepoint();
}

void example() {
  jl_value_t *slot = NULL;
  JL_GC_PUSH1(&slot);
  jl_do_processing(&slot);
  JL_GC_POP();
}
```

JL_GLOBALLY_ROOTED

This annotation implies that a given value is always globally rooted. It can be applied to global variable declarations, in which case it will apply to the value of those variables (or values if the declaration if for an array), or to functions, in which case it will apply to the return value of such functions (e.g. for functions that always return some private, globally rooted value).

Usage example:

```
extern JL_DLLEXPORT jl_datatype_t *jl_any_type JL_GLOBALLY_ROOTED;
jl_ast_context_t *jl_ast_ctx(fl_context_t *fl) JL_GLOBALLY_ROOTED;
```

JL_ALWAYS_LEAFTYPE

This annotations is essentially equivalent to `JL_GLOBALLY_ROOTED`, except that is should only be used if those values are globally rooted by virtue of being a leaftype. The rooting of leaftypes is a bit complicated. They are generally rooted through `cache` field of the corresponding `TypeName`, which itself is rooted by the containing module (so they're rooted as long as the containing module is ok) and we can generally assume that leaftypes are rooted where they are used, but we may refine this property in the future, so the separate annotation helps split out the reason for being globally rooted.

The analyzer also automatically detects checks for leaftype-ness and will not complain about missing GC roots on these paths.

```
JL_DLLEXPORT jl_value_t *jl_apply_array_type(jl_value_t *type, size_t dim) JL_ALWAYS_
```

JL_GC_PROMISE_ROOTED

This is a function-like annotation. Any value passed to this annotation will be considered rooted for the scope of the current function. It is designed as an escape hatch for analyzer inadequacy or complicated situations. However, it should be used sparingly, in favor of improving the analyzer itself.

```
void example() {
  jl_value_t *val = jl_alloc_something();
  if (some_condition) {
    // We happen to know for complicated external reasons
    // that val is rooted under these conditions
    JL_GC_PROMISE_ROOTED(val);
  }
}
```

Completeness of analysis

The analyzer only looks at local information. In particular, e.g. in the `PROPAGATES_ROOT` case above, it assumes that such memory is only modified in ways it can see, not in any called functions (unless it happens to decide to consider them in its analysis) and not in any concurrently running threads. As such, it may miss a few problematic cases, though in practice such concurrent modification is fairly rare. Improving the analyzer to handle more such cases may be an interesting topic for future work.

« Julia SSA-form IR Reporting and analyzing crashes (segfaults) »