orphan:

Remote mirrors proposal

Contents

This proposal describes a new implementation for nominal type metadata which will enable out-of-process heap inspection, intended for use by debugging tools for detecting leaks and cycles. This implementation will subsume the existing reflection support for Mirrors, enabling out-of-process usage while also reducing generated binary size.

Radars tracking this work:

- rdar://problem/15617914
- rdar://problem/17019505
- rdar://problem/20771693

Goals and non-goals

We wish to do post-mortem debugging of memory allocations in a Swift program. Debugging tools can already introspect the memory allocator to identify all live memory allocations in the program's heap.

If the compiler were to emit the necessary metadata, the layout of most allocations can be ascertained, and in particular we can identify any references inside the heap object. This metadata can be used together with the core dump of a program to build a graph of objects.

We have to be able to get all the necessary information without executing any code in the address space of the target, since it may be dead or otherwise in a funny state.

In order to identify strong retain cycles, we need to know for each reference if it is strong, weak, or unowned.

We wish to be able to opt out of metadata selectively. For secrecy, we might want to strip out field names, but keep metadata about which fields contain references. For release builds, we might want to strip out most of the field metadata altogether, except where explicitly required for code that relies on reflection for functionality.

It would be better if the new functionality subsumes some of the existing metadata, instead of adding a whole new set of structures that the compiler and runtime must keep in sync.

While this should have zero runtime overhead when not in use, it is OK if introspection requires some additional computation, especially if it can be front-loaded or memoized.

It is OK if in rare cases the metadata is not precise -- for some allocations, we might not be able to figure out the runtime type of the contained value. Also we will not attempt to verify the "roots" of the object graph by walking the stack for pointers.

Types of heap allocations

There are several types of heap allocations in Swift. We mostly concern ourselves with class instances for now, but in the fullness of time we would like to have accurate metadata for all heap allocations.

Swift class instances

These have an isa pointer that points to a class metadata record.

Objective-C class instances

These also have an isa pointer, but the class metadata record has the Objective-C bit set.

Boxes

These are used for heap-allocating mutable values captured in closures, for indirect enum cases, and for Error existential values. They have an identifying isa pointer and reference count, but the isa pointer is shared by all boxes and thus does not describe the heap layout of the box.

Contexts

The context for a thick function is laid out like a tuple consisting of the captured values. Currently, the only aspect of the layout that is needed by the runtime is knowledge of which captured values are heap pointers. A unique isa pointer is created for each possible layout here.

Blocks

Blocks are similar to contexts but have a common header and package the function pointer and captured values in a single retainable heap object.

Metatypes

Runtime-allocated metatypes will appear in the malloc heap. They themselves cannot contain heap references though.

Opaque value buffers

These come up when a value is too large to fit inside of an existential's inline storage, for example. They do not have a header, so we will not attempt to introspect them at first -- eventually, we could identify pointers to buffers where the existential is itself inside of a heap-allocated object.

Existing metadata

Swift already has a lot of reflective features and much of the groundwork for this exists in some form or another, but each one is lacking in at least one important respect.

Generic type metadata

The isa pointer of an object points to a metadata record. For instances of generic class types, the metadata is lazily instantiated from the generic metadata template together with the concrete types that are bound to generic parameters.

Generic type metadata is instantiated for generic classes with live instances, and for metatype records of value types which are explicitly referenced from source.

When the compiler needs to emit a generic type metadata record, it uses one of several strategies depending on the type being referenced. For concrete non-generic types, a direct call to a lazy accessor can be generated. For bound generic types T<P1, ..., Pn>, we recursively emit metadata references for the generic parameters Pn, then call the getter for the bound type T. For archetypes -- that is, generic type parameters which are free variables in the function body being compiled -- the metadata is passed in as a value, so the compiler simply emits a copy of that.

Generic type metadata tells us the size of each heap allocation, but does not by itself tell us the types of the fields or what references they contain.

Mirrors and NominalTypeDescriptors

The implementation of Mirrors uses runtime primitives which introspect the fields of an opaque value by looking at the NominalTypeDescriptor embedded in a type's metadata record.

For structures and classes, the NominalTypeDescriptor contains a function pointer which returns an array of field types. The function pointer points to a "field type metadata function" emitted by the compiler. This function emits metadata record references for each field type and collects them in an array. Since the isa pointer of a class instance points at an instantiated type, the field types of such a NominalTypeDescriptor are also all concrete types.

NominalTypeDescriptors record field names, in addition to types. Right now, all of this information is stored together, without any way of stripping it out. Also, NominalTypeDescriptors do not record whether a reference is strong, weak or unowned, but that would be simple to fix.

A bigger problem is that we have to call a function to lazily generate the field type metadata. While a NominalTypeDescriptor for every instantiated class type appears in a crashed process, the field types do not, because only a call to the field type function will instantiate them.

Objective-C instance variable metadata

The Objective-C runtime keeps track of the types of instance variables of classes, and there is enough information here to identify pointers in instances of concrete types, however there's no support for generic types. We could have generic type metadata instantiation also clone and fill in templates for Objective-C instance variables, but this would add a runtime cost to a feature that is primarily intended for debugging.

DWARF metadata

IRGen emits some minimal amount of DWARF metadata for non-generic types, but makes no attempt to describe generic type layout to the debugger in this manner.

However, DWARF has the advantage that it can be introspected without running code, and stripped out.

New field type metadata format

The main limitation of all of the above is either an inability to reason about generic types, or the requirement to run code in the target.

Suppose T is a generic type, and S is some set of substitutions.

The compiler conceptually implements an operation G(T, S) which returns a lazily-instantiated type descriptor for the given input parameters. However, its really performing a partial evaluation G(T)(S), with the "G(T)" part happening at compile time.

Similarly, we can think of the field type access function as an operation F(T, S) which returns the types of the fields of T, with T again fixed at compile time.

What we really want here is to build an "interpreter" -- or really, a parser for a simple serialized graph -- which understands how to parse uninstantiated generic metadata, keep track of substitutions, and calculate field offsets, sizes, and locations of references.

This "interpreter" has to be able to find metadata for leaf types "from scratch", and calculate field sizes and offsets in the same way that generic type metadata instantiation calculates object sizes.

The "interpreter" will take the form of a library for understanding field type metadata records and symbolic type references. This will be a C++ library and it needs to support the following use cases:

1. In-process reflection, for backing the current Mirrors in the standard library
2. Out-of-process reflection, for heap debugging tools
3. Out-of-process reflection, for a new remote Mirrors feature in the library (optional)

The API will be somewhat similar to Mirrors as they are in the stdlib today.

The details are described below.

Symbolic type references

Since we're operating on uninstantiated generic metadata, we need some way to describe compositions of types. Instead of using metadata record pointers, which are now insufficient, we use type references written in a mini-language.

A symbolic type reference is a recursive structure describing an arbitrary Swift AST type in terms of nominal types, generic type parameters, and compositions of them, such as tuple types.

For each AST type, we can distinguish between the minimum information we need to identify heap references therein, and the full type for reflection. The former could be retained while the latter could be stripped out in certain builds.

We already have a very similar encoding -- parameter type mangling in SIL. It would be good to re-use this encoding, but for completeness, the full format of a type reference is described below:

1. **A built-in type reference.** Special tokens can be used to refer to various built-in types that have runtime support.
2. **A concrete type reference.** This can either be a mangled name of a type, or a GOT offset in the target.
3. **A heap reference.** This consists of:
   - strong, weak or unowned
   - (optional) a reference to the class type itself
4. **A bound generic type.** This consists of:
   - A concrete or built-in type reference
   - A nested symbolic type reference for each generic parameter
5. **A tuple type.** This consists of:
   - A recursive sequence of symbolic type references.
6. **A function type.** This consists of:
   - A representation,
   - (optional) input and output types
7. **A protocol composition type.** This consists of:
   - A flag indicating if any of the protocols are class-constrained, which changes the representation
   - The number of non-@objc protocols in the composition
   - (optional) references to all protocols in the composition
8. **A metatype.** This consists of:
   - (optional) a type reference to the instance type

    - there's no required information -- a metatype is always a single pointer to a heap object which itself does not reference any other heap objects.
9. **An existential metatype.** This consists of:
    - The number of protocols in the composition.
    - (optional) type references to the protocol members.
10. **A generic parameter.** Within the field types of a generic type, references to generic parameters can appear. Generic parameters are uniquely identifiable by an index here (and once we add nested generic types, a depth).

You can visualize type references as if they are written in an S-expression format -- but in reality, it would be serialized in a compact binary form:

```
(tuple_type
  (bound_generic_type
    (concrete_type "Array")
    (concrete_type "Int"))
  (bound_generic_type
    (builtin_type "Optional")
    (generic_type_parameter_type index=0)))
```

We will provide a library of standalone routines for decoding, encoding and manipulating symbolic type references.

Field type metadata records

We introduce a new type of metadata, stored in its own section so that it can be stripped out, called "field type metadata". For each nominal type, we emit a record containing the following:

1. the name of the nominal type,
2. the number of generic parameters,
3. type references, written in the mini-language above, for each of its field types.
4. field names, if enabled.

Field type metadata is linked together so that it can be looked up by name, post-mortem by introspecting the core dump.

We add a new field to the NominalTypeDescriptor to store a pointer to field type metadata for this nominal type. In "new-style" NominalTypeDescriptors that contain this field, the existing field type function will point to a common field type function, defined in the runtime, which instantiates the field type metadata. This allows for backward compatibility with old code, if desired.

Field type metadata instantiation

First, given an isa pointer in the target, we need to build the symbolic type reference by walking backwards from instantiated to uninstantiated metadata, collecting generic parameters. This operation is lazy, caching the result for each isa pointer.

```
enum SymbolicTypeReference {
  case Concrete(String)
  case BoundGeneric(String, [SymbolicTypeReference])
  case Tuple([SymbolicTypeReference])
  ...
}

func getSymbolicTypeOfObject(_ isa: void*) -> SymbolicTypeReference
```

Next, we define an "instantiation" operation, which takes a completely substituted symbolic type reference, and returns a list of concrete field types and offsets.

This operation will need to recursively visit field metadata records and keep track of generic parameter substitutions in order to correctly calculate all field offsets and sizes.

The result of instantiating metadata for each given SymbolicTypeReference can be cached for faster lookup.

This library has to be careful when following any pointers in the target, to properly handle partially-initialized objects, runtime bugs that led to memory corruption, or malicious code, without crashing or exploiting the debugging tools.

```
enum FieldLayout {
  // the field contains a heap reference
  case Strong, Weak, Unowned
  // the field is an opaque binary blob, contents unknown.
  case Opaque
  // the field is a value type -- look inside recursively.
  case ValueType(indirect field: FieldDescriptor)
}

struct FieldDescriptor {
  let size: UInt
  let align: UInt
  let offset: UInt
  let layout: FieldLayout
}
```

```
func instantiateSymbolicType(_ ref: SymbolicTypeReference) -> [FieldTypeDescriptor]
```

Field type metadata can have circular references -- for example, consider two classes which contain optionals of each other. In order to calculate field offsets correctly, we need to break cycles when we know something is a class type, and use a work-list algorithm instead of unbounded recursion to ensure forward progress.

Enum type metadata

For enums, the field metadata record will also need to contain enough information about the spare bits and tag bits of the payload types that we can at runtime determine the case of an enum and project the payload, again without running code in the target.

This will allow us to remove a pair of value witness functions generated purely for reflection, since they don't seem to be performance-critical.

Closures

For closure contexts and blocks, it would be nice to emit metadata, too.

Secrecy and release builds

There are several levels of metadata we can choose to emit here:

1. For code that requires runtime for functional purposes, or for the standard library in debug builds, we can have a protocol conformance or compiler flag enable unconditional emission of all metadata.
2. For system frameworks, we can omit field names and replace class names with unique identifiers, but keep the type metadata to help users debug memory leaks where framework classes are retaining instances of user classes.
3. For release builds, we can strip out all the metadata except where explicitly required in 1).

This probably requires putting the required metadata in a different section from the debug metadata. Perhaps field names should be separate from symbolic type references too.

Performance

Since the field type metadata instantiation only happens once per isa pointer, mirrors will not suffer a performance impact beyond the initial warm-up time. Once the field type descriptor has been constructed, reflective access of fields will proceed as before.

There might also be a marginal performance gain from removing all the field type functions from the text segment, where they're currently interspersed with other code, and replacing them with read only data containing no relocations, which won't get paged in until needed.

Resilience

We may choose to implement the new metadata facility after stabilizing the ABI. In this case, we should front-load some engineering work on NominalTypeDescriptors first, to make them more amenable to future extension.

We need to carefully review the new metadata format and make sure it is flexible enough to support future language features, such as bound generic existentials, which may further complicate heap layout.

As described above, it is possible to introduce this change in a backwards-compatible manner. We keep the field type function field in the NominalTypeDescriptor, but for "new-style" records, set it to point to a common function, defined in the runtime, which parses the new metadata and returns an array of field types that can be used by old clients.

Testing

By transitioning mirrors to use the new metadata, existing tests can be used to verify behavior. Additional tests can be developed to perform various allocations and assert properties of the resulting object graph, either from in- or out-of-process.

If we go with the gradual approach where we have both field type functions and field type metadata, we can also instantiate the former and compare it against the result of invoking the latter, for all types in the system, as a means of validating the field type metadata.