SIL utilities for modeling memory access

The `AccessBase`, `AccessStorage` and `AccessPath` types formalize memory access in SIL. Given an address-typed SIL value, it is possible to reliably identify the storage location of the accessed memory. `AccessStorage` identifies an accessed storage location. `AccessPath` contains both a storage location and the "access path" within that memory object. The relevant API details are documented in MemAccessUtils.h

Formal access

SIL preserves the language semantics of formal variable access in the form of access markers. `begin_access` identifies the address of the formal access and `end_access` delimits the scope of the access. At the language level, a formal access is an access to a local variable or class property. For details, see SE-0176: Enforce Exclusive Access to Memory

Access markers are preserved in SIL to:

1. verify exclusivity enforcement
2. optimize exclusivity checks following other transforms, such as converting dynamic checks into static checks
3. simplify and strengthen general analyses of memory access. For example, `begin_access [read] %address` indicates that the accessed address is immutable for the duration of its access scope

Access path def-use relationship

Computing `AccessStorage` and `AccessPath` for any given SIL address involves a use-def traversal to determine the origin of the address. It may traverse operations on values of type address, Builtin.RawPointer, box, and reference. The logic that formalizes which SIL operations may be involved in the def-use chain is encapsulated with the `AccessUseDefChainVisitor`. The traversal can be customized by implementing this visitor. Customization is not expected to change the meaning of AccessStorage or AccessPath. Rather, it is intended for additional pass-specific book-keeping or for higher-level convenience APIs that operate on the use-def chain bypassing AccessStorage completely.

Access def-use chains are divided by four points: the object "root", the access "base", the outer-most "access" scope, and the "address" of a memory operation. For example:

```
struct S {
  var field: Int64
}
class C {
  var prop: S
}

%root    = alloc_ref $C
%base    = ref_element_addr %root : $C, #C.prop
%access  = begin_access [read] [static] %base : $*S
%address = struct_element_addr %access : $*S, #.field
%value   = load [trivial] %address : $*Int64
end_access %access : $*S
```

OR

```
  %root    = alloc_box $S
  %base    = project_box %root : ${ var S }
  %access  = begin_access [read] [static] %base : $*S
  %address = struct_element_addr %access : $*S, #.field
  %value   = load [trivial] %address : $*Int64
  end_access %access : $*S
```

Reference root

The first part of the def-use chain computes the formal access base from the root of the object (e.g. `alloc_ref -> ref_element_addr` and `alloc_box -> project_box`). The reference root might be a locally allocated object, a function argument, a function result, or a reference loaded from storage. There is no enforcement on the type of operation that can produce a reference; however, only reference types, Builtin.BridgeObject types, and box types are allowed in this part of the def-use chain. The reference root is the greatest common ancestor in the def-use graph that can identify an object by a single SILValue. If the root is an `alloc_ref`, then it is *uniquely identified*. The def-use chain from the root to the base may contain reference casts (`isRCIdentityPreservingCast`) and phis.

This example has an identifiable def-use chain from `%root` to `%base`:

```
class A {
  var prop0: Int64
}
class B : A {
}

bb0:
  %root = alloc_ref $B
  cond_br _, bb1, bb2

bb1:
  %a1 = upcast %root : $B to $A
  br bb3(%a1 : $A)

bb2:
  %a2 = upcast %root : $B to $A
  br bb3(%a2 : $A)

bb3(%a : $A):
  %bridge = ref_to_bridge_object %a : $A, %bits : $Builtin.Word
  %ref = bridge_object_to_ref %bridge : $Builtin.BridgeObject to $A
  %base = ref_element_addr %ref : $A, #A.prop0
```

Each object property and its tail storage is considered a separate formal access base. The reference root is only one component of an `AccessStorage` location. AccessStorage also identifies the class property being accessed within that object.

A reference root may be borrowed, so the use-def path from the base to the root may cross a borrow scope. This means that uses of one base may not be replaced with a different base even if it has the same AccessStorage because they may not be contained within the same borrow scope. However, this is the only part of the access path that may be borrowed. Address uses with the same base can be substituted without checking the borrow scope.

Access base

The access base is the address or Builtin.RawPointer type SILValue produced by an instruction that directly identifies the kind of storage being accessed without further use-def traversal. Common access bases are `alloc_stack`, `global_addr`, `ref_element_addr`, `project_box`, and function arguments (see `AccessStorage::Kind`).

The access base is the same as the "root" SILValue for all storage kinds except global and reference storage. Reference storage includes class, tail and box storage. Global storage has no root. For reference storage the root is the SILValue that identifies object, described as the "reference root" above.

"Box" storage is uniquely identified by an `alloc_box` instruction. Therefore, we consider the `alloc_box` to be the base of the access. Box storage does not apply to all box types or box projections, which may instead originate from arguments or indirect enums for example.

An access scope, identified by a `begin_access` marker, may only occur on the def-use path between the access base and any address projections. The def-use path from the root to the base cannot cross an access scope. Likewise, the def-use between an access projection and the memory operation cannot cross an access scope.

Typically, the base is the address-type source operand of a `begin_access`. However, the path from the access base to the `begin_access` may include *storage casts* (see `isAccessStorageCast`). It may involve address an pointer types, and may traverse phis. For some kinds of storage, the base may itself even be a non-address pointer. For phis that cannot be uniquely resolved, the base may even be a box type.

This example has an identifiable def-use chain from `%base` to `%access`:

```
bb0:
  %base = alloc_box $Int { var Int }
  %boxadr = project_box %base : ${ var Int }
  %p0 = address_to_pointer %boxadr : $*Int to $Builtin.RawPointer
  cond_br _, bb1, bb2

bb1:
  %p1 = copy_value %p0 : $Builtin.RawPointer
  br bb3(%p1 : $Builtin.RawPointer)

bb2:
  br bb3(%p0 : $Builtin.RawPointer)

bb3(%ptr : $Builtin.RawPointer):
  %adr = pointer_to_address %ptr : $Builtin.RawPointer to $*Int
  %access = begin_access [read] [static] %adr : $*Int
```

Note that address-type phis are illegal (full enforcement pending). This is important for simplicity and efficiency, but also allows for a class of storage optimizations, such as bitfields, in which address storage is always uniquely determined. Currently, if a (non-address) phi on the access path from `base` to `access` does not have a common base, then it is considered an invalid access (the AccessStorage object is not valid). SIL verification ensures that a formal access always has valid AccessStorage (WIP). In other words, the source of a `begin_access` marker must be a single, non-phi base. In the future, for further simplicity, we may also disallow pointer phis unless they have a common base.

Not all SIL memory access is part of a formal access, but the `AccessStorage` and `AccessPath` abstractions are universally applicable. Non-formal access still has an access base, even though the use-def search does not begin at a `begin_access` marker. For non-formal access, SIL verification is not as strict. An invalid access is allowed, but handled conservatively. This is safe as long as those non-formal accesses can never alias with class and global storage. Class and global access must always be guarded by formal access markers--at least until static markers are stripped from SIL.

Nested access

Nested access occurs when an access base is a function argument. The caller always checks `@inout` arguments for exclusivity (an access marker must exist in the caller). However, the argument itself is a variable with its own formal access. Conflicts may occur in the callee which were not evident in the caller. In this example, a conflict occurs in `hasNestedAccess` but not in its caller:

```
func takesTwoInouts(_ : inout Int, _ : inout Int) -> () {}

func hasNestedAccess(_ x : inout Int) -> () {
  takesTwoInouts(&x, &x)
}

var x = 0
hasNestedAccess(&x)
```

Produces these access markers:

```
sil @takesTwoInouts : $@convention(thin) (@inout Int, @inout Int) -> ()

sil @hasNestedAccess : $@convention(thin) (@inout Int) -> () {
bb0(%0 : $*Int):
  %innerAccess  = begin_access [modify] %0 : $*Int
  %conflicting  = begin_access [modify] %0 : $*Int
  %f = function_ref @takesTwoInouts
  apply %f(%innerAccess, %conflicting)
    : $@convention(thin) (@inout Int, @inout Int) -> ()
  end_access %conflicting : $*Int
  end_access %innerAccess : $*Int
  //...
}

%var = alloc_stack $Int
%outerAccess  = begin_access [modify] %var : $*Int
%f = function_ref @hasNestedAccess
apply %f(%outerAccess) : $@convention(thin) (@inout Int) -> () {
end_access %outerAccess : $*Int
```

Nested accesses become part if the def-use chain after inlining. Here, both `%innerAccess` and `%conflicting` are nested within `%outerAccess`:

```
%var = alloc_stack $Int
%outerAccess  = begin_access [modify] %var : $*Int
%innerAccess  = begin_access [modify] %outerAccess : $*Int
%conflicting  = begin_access [modify] %outerAccess : $*Int
```

```
%f = function_ref @takesTwoInouts
apply %f(%innerAccess, %conflicting)
  : $@convention(thin) (@inout Int, @inout Int) -> ()
end_access %conflicting : $*Int
end_access %innerAccess : $*Int
end_access %outerAccess : $*Int
```

For most purposes, the inner access scopes are irrelevant. When we ask for the "accessed storage" for `%innerAccess`, we get an `AccessStorage` value of "Stack" kind with base `%var = alloc_stack`. If instead of finding the original accessed storage, we want to identify the enclosing formal access scope, we need to use a different API that supports the special `Nested` storage kind. This is typically only used for exclusivity diagnostics though.

TODO: Nested static accesses that result from inlining could potentially be removed, as long as DiagnoseStaticExclusivity has already run.

Access projections

On the def-use chain between the *outermost* formal access scope within the current function and a memory operation, *access projections* identify subobjects laid out within the formally accessed variable. The sequence of access projections between the base and the memory address correspond to an access path.

For example, there is no formal access for struct fields. Instead, they are addressed using a `struct_element_addr` within the access scope:

```
%access  = begin_access [read] [static] %base : $*S
%memaddr = struct_element_addr %access : $*S, #.field
%value   = load [trivial] %memaddr : $*Int64
end_access %access : $*S
```

Note that is is possible to have a nested access scope on the address of a struct field, which may show up as an access of struct_element_addr after inlining. The rule is that access projections cannot occur outside of the outermost access scope within the function.

Access projections are address projections--they take an address at operand zero and produce a single address result. Other straightforward access projections include `tuple_element_addr`, `index_addr`, and `tail_addr` (an aligned form of `index_addr`).

Enum payload extraction (`unchecked_take_enum_data_addr`) is also an access projection, but it has no effect on the access path.

Indirect enum payload extraction is a special two-instruction form of address projection (`load : ${ var } -> project_box`). For simplicity, and to avoid the appearance of box types on the access path, this should eventually be encapsulated in a single SIL instruction.

For example, the following complex def-use chain from `%base` to `%load` actually has an empty access path:

```
%boxadr = unchecked_take_enum_data_addr %base : $*Enum<T>, #Enum.int!enumelt
%box = load [take] %boxadr : $*<τ_0_0> { var Int } <T>
%valadr = project_box %box : $<τ_0_0> { var Int } <T>, 0
%load = load [trivial] %valadr : $*Int
```

Storage casts may also occur within an access. This typically results from accessors, which perform address-to-pointer conversion. Pointer-to-address conversion performs a type cast, and could lead to different subobject types corresponding to the same base and access path. Access paths still uniquely identify a memory location because it is illegal to cast memory to non-layout-compatible types on same execution path (without an intervening `bind_memory`).

Address-type phis are prohibited, but because pointer and box types may be on the def-use chain, phis may also occur on an access path. A phi is only a valid part of an access path if it has no affect on the path components. This means that pointer casting and unboxing may occur on distinct phi paths, but index offsets and subobject projections may not. These rules are currently enforced to a limited extent, so it's possible for invalid access path to occur under certain conditions.

For example, the following is a valid def-use access chain, with an access base defined in `bb0`, a memory operation in `bb3` and an `index_addr` and `struct_element_addr` on the access path:

```
class A {}

struct S {
  var field0: Int64
  var field1: Int64
}

bb0:
  %base    = ref_tail_addr %ref : $A, $S
  %idxproj = index_addr %tail : $*S, %idx : $Builtin.Word
  %p0 = address_to_pointer %idxproj : $*S to $Builtin.RawPointer
  cond_br _, bb1, bb2

bb1:
  %pcopy = copy_value %p0 : $Builtin.RawPointer
  %adr1  = pointer_to_address [strict] %pcopy : $Builtin.RawPointer to $*S
  %p1    = address_to_pointer %adr1 : $*S to $Builtin.RawPointer
  br bb3(%p1 : $Builtin.RawPointer)

bb2:
  br bb3(%p0 : $Builtin.RawPointer)

bb3(%p3 : $Builtin.RawPointer):
  %adr3 = pointer_to_address [strict] %p3 : $Builtin.RawPointer to $*S
  %field = struct_element_addr %adr3 : $*S, $S.field0
  load %field : $*Int64
```

AccessStorage

`AccessStorage` identifies an accessed storage location, be it a box, stack location, class property, global variable, or argument. It is implemented as a value object that requires no compile-time memory allocation and can be used as the hash key for that location. Extra bits are also available for information specific to a particular optimization pass. Its API provides the kind of location being accessed and information about the location's uniqueness or whether it is distinct from other storage.

Two **uniquely identified** storage locations may only alias if their AccessStorage objects are identical.

`AccessStorage` records the "root" SILValue of the access. The root is the same as the access base for all storage kinds except global and class storage. For class properties, the storage root is the reference root of the object, not the base of the property. Multiple `ref_element_addr` projections may exist for the same property. Global variable storage is always uniquely identified, but it is impossible to find all uses from the def-use chain alone. Multiple `global_addr` instructions may reference the same variable. To find all global uses, the client must independently find all global variable references within the function. Clients that need to know which SILValue base was discovered during use-def traversal in all cases can make use of `AccessStorageWithBase` or `AccessPathWithBase`.

AccessPath

`AccessPath` extends `AccessStorage` to include the path components that determine the address of a subobject within the access base. The access path is a string of index offsets and subobject projection indices.

```
struct S {
  var field0: Int64
  var field1: Int64
}

%eltadr = struct_element_addr %access : $*S, #.field1

Path: (#1)
```

```
class A {}

%tail  = ref_tail_addr %ref : $A, $S
%one   = integer_literal $Builtin.Word, 1
%elt   = index_addr %tail : $*S, %one : $Builtin.Word
%field = struct_element_addr %elt : $*S, $S.field0

Path: (@1, #0)
```

Note that a projection from a reference type to the object's property or tail storage is not part of the access path because it is already identified by the storage location.

Offset indices are all folded into a single index at the head of the path (a missing offset implies offset zero). Offsets that are not static constants are still valid but are labeled "@Unknown". Indexing within a subobject is an ill-formed access, but is handled conservatively since this rule cannot be fully enforced.

For example, the following is an invalid access path, which just happens to point to field1:

```
%field0 = struct_element_addr %base : $*S, #field0
%field1 = index_addr %elt : $*Int64, %one : $Builtin.Word

Path: (INVALID)
```

The following APIs determine whether an access path contains another or may overlap with another.

`AccessPath::contains(AccessPath subPath)`

`AccessPath::mayOverlap(AccessPath otherPath)`

These are extremely light-weight APIs that, in the worst case, require a trivial linked list traversal with single pointer comparison for the length of subPath or otherPath.

Subobjects are both contained with and overlap with their parent storage. An unknown offset does not contain any known offsets but overlaps with all offsets.

Access path uses

For any accessed storage location and base, it must also be possible to reliably identify all uses of that storage location within the function for a particular access base. If the storage is uniquely identified, then that also implies that all uses of that storage within the function have been discovered. In other words, there are no aliases to the same storage that aren't covered by this use set.

The `AccessPath::collectUses()` API does this. It is possible to ask for only the uses contained by the current path, or for all potentially overlapping uses. It is guaranteed to return a complete use set unless the client specifies a limit on the number of uses.

As passes begin to adopt AccessPath::collectUses(), I expect it to become a visitor pattern that allows the pass to perform custom book-keeping for certain types of uses.

The `AccessPathVerification` pass runs at key points in the pipeline to ensure that all address uses are identified and have consistent access paths. This pass ensures that the implementations of AccessPath is internally consistent for all SIL patterns. Enforcing the validity of the SIL itself, such as which operations are allowed on an access def-use chain, is handled within the SIL verifier instead.