!standard 3.10(14/3) 18-01-18 AI12-0240-1/08
!class Amendment 17-10-09
!status work item 17-10-09
!status received 17-10-05
!priority Very_Low
!difficulty Hard
!subject Access value ownership and parameter aliasing
!summary

** TBD.

!problem

Pointers (access types) are essential to many complex Ada data structures, but they also have significant downsides, and can create many kinds of safety and security problems. The question is can we create a subset of access-type functionality which supports the creation of interesting data structures without bringing all of the downsides. The notion of pointer "ownership" has emerged as one way to "tame" pointer problems, while preserving flexibility. The goal is to allow a pattern of use of pointers that avoids dangling references as well as storage leaks, by providing safe, immediate, automatic reclamation of storage rather than relying on unchecked deallocation, while also not having to fall back on the time and space vagaries of garbage collection. As a side benefit, we can also get safer use of pointers in the context of parallelism. This AI proposes the use of pointer ownership, as well as some additional rules preventing "aliasing" between parameters, and between parameters and global variables, to provide safe, automatic, parallelism-friendly heap storage management while allowing the flexible construction of pointer-based data structures, such as trees, linked lists, hash tables, etc.

When attempting to prove properties about a program, particularly programs with multiple threads of control, the enemy is often the unknown "aliasing" of names introduced by access types and certain uses of (potentially) by-reference parameters. By unknown aliasing of names, we mean the situation where two distinct names might refer to the same object, without the compiler being aware of that. A rename introduces an alias, but not an "unknown alias," because the compiler is fully aware of such an alias, and hence is not something we are worrying about here. On the other hand, if a global variable is passed by reference as a parameter to a subprogram that also has visibility on the same global, the by-reference parameter and the global are now aliases within the subprogram, and the compiler generating code for the subprogram has no way of knowing that, hence they are "unknown" aliases. One approach is to always assume the worst, but that makes analysis much harder, and in some cases infeasible.

Access types also introduce unknown aliasing, and in most cases, an analysis tool will not be sure whether the aliases exist, and will again have to make worst-case assumptions, which

again may make any interesting proof infeasible.

!proposal

This AI introduces a restriction against parameter aliasing (No_Parameter_Aliasing), and an aspect called "Ownership" for access types and composite types, which together provide safe, automatic storage management. Furthermore this set of features can reduce potential aliasing to the point that the only remaining potential aliases involve names denoting elements or slices of the same array using dynamic indices. This sort of aliasing is felt to be manageable by most analysis tools, as it is restricted to determining the possibility of numeric equality between such indices. Run-time checks are defined to deal with this sort of aliasing.

Pointer ownership enables automatic storage reclamation since there is never more than one “owning” pointer to a given object, and the pointers are generally limited to pointing only at heap objects (never at a stack-allocated object or aliased component). This restriction matches that provided by "pool-specific" access types, so in this AI we limit the notion of pointer ownership for named access types to such types. We could incorporate a restricted set of "general" access types as well, but that would require more rules, and in this version we choose to go with the simpler, pool-specific-only, approach. We do permit use of certain objects of an anonymous access type, but these have restricted use, and in any case cannot be used with Unchecked_Deallocation.

Access type Ownership:

If a pool-specific access-to-variable type has the value True for the aspect Ownership, then certain operations on the access type are restricted, and certain operations result in an automatic deallocation of the designated object. The Ownership aspect is inherited by derived types, and is non-overridable. On a root type, the Ownership aspect defaults to False. We also define the aspect on composite types and private types.

The basic rule is that at most one object of such an access-to-variable type with Ownership True may be used to refer to a designated object at any time. There might be multiple access-to-variable (or access-to-constant) objects that designate the same object (or some part of it) at any given time, but at most one of them may be dereferenced. All of the others are considered "invalid," because their value has been "borrowed" (see further below).

Conversely, there might be multiple "valid" access-to-constant objects that designate a given object, so long as all access-to-variable objects designating the object are at that point invalid (or "observed" -- see below). This is the usual "single writer, multiple reader" rule for safe access to a shared variable.

We say an access-to-variable object is "observed" when its value has been "observed", and such an object can only be used for read-only access to the designated object.

To summarize the possible states of an access-to-variable object of a type with pointer ownership (changing "valid" to "unrestricted" below):

1) "unrestricted" -- may be dereferenced and used to update the designated object.
2) "observed" -- value has been "observed," meaning that other access-to-constant objects or IN formal parameters may exist that give read-only access to all or part of the designated object, or part of some object directly or indirectly owned by the designated object.
3) "borrowed" -- value has been "borrowed," meaning that the value has been copied into a short-lived access-to-variable object, such as an IN parameter, possibly of an anonymous access type. "Borrowed" also covers the case where some part of the designated object is passed as a potentially by-ref [IN] OUT parameter. Finally, it also covers the case where a subcomponent that is itself an access-to-variable with ownership, is borrowed. The state reverts to unrestricted when the short-lived reference no longer exists.

An access-to-variable object of a named type that is unrestricted or observed is considered to be the "owner" of its designated object. There should be exactly one owner of every existing object created by an allocator for an access type with a True Ownership aspect. While observed, the access object is itself read-only. When an object is borrowed, ownership has been temporarily transferred to another object, and while in such a state the original access object is not usable at all.

Ownership is transitive, in that if a pointer owns a designated object, it indirectly owns every object owned by one of its access subcomponents.

To simplify the rules, we introduce the notion of *managed* objects, which come in two varieties:

1. *Ownership Objects* -- Objects of a type with Ownership aspect True:
    a. *Owning access objects* -- access-to-variable objects with Ownership aspect True
    b. *Observing access objects* -- access-to-constant objects with Ownership aspect True
    c. Composite Ownership Objects
2. *Other Managed Objects* -- Any part of a non-Ownership object designated by an owning (1.a above) or observing access object (1b. above)

We require pass-by-reference for all types with Ownership aspect True, unless they are visibly an access type. This is important to properly deal with the possibility of exceptions, because pass-by-copy for [**in**] **out** composite parameters could produce dangling references if a subprogram propagated an exception that was then caught. (Ownership aspect is not defined at all for scalar types, and we don't allow a private type with Ownership aspect True to be of a by-copy type.)

Note that the rules are actually expressed in terms of “names” rather than “objects” since

objects don't really exist at compile-time. In the rules, we distinguish "static" names from "dynamic" names, where static names are those made of names denoting stand-alone objects, dereferences of owning/observing access objects, and selected components, while "dynamic" names can involve indexing or slicing.

!wording

Add the following sections to Annex H:

H.7 The Ownership Aspect

This section describes the Ownership aspect, which can be specified for an access type so that there can be either one writable access path to an object designated by an object of such an access type, or one or more read-only access paths via such access objects, but never both. Furthermore, objects designated by such access objects are automatically finalized and deallocated when the last access path is removed.

Static Semantics

The following aspect may be specified for a named access-to-object type, a stand-alone object (including a generic formal **in**-mode object) of an anonymous access-to-object type, or a composite type:

Ownership

> The Ownership aspect is of type Boolean, and shall be specified by a static expression. The Ownership aspect for a named general access type is False, and only confirming specifications are permitted. The Ownership aspect is nonoverridable for a pool-specific access type, and if not specified for a root pool-specific access type, the value is False. If not specified for a stand-alone object (or generic formal **in**-mode object) of an anonymous access-to-object type, the value is False if there is no initialization (or default) expression, and otherwise it is the same as that of the type of the initialization (or default) expression. For an untagged composite type, the Ownership aspect is nonoverridable in derived types. For a nonlimited controlled type, the Ownership aspect is False, and only confirming specifications are permitted. For other composite types, the Ownership aspect is True if there are any visible subcomponents for which the Ownership aspect is True or if the type is tagged and its parent (if any) has Ownership aspect True, and in these cases only confirming specifications are permitted.
>
> > AARM Ramification: We do not require that all descendants of a tagged type with Ownership aspect False also have Ownership False, but we disallow (implicit or explicit) conversion between types with differing Ownership, preventing inheritance of dispatching operations across such an Ownership aspect change.
> > SPARK Note: We could probably allow inheriting dispatching operations in

SPARK, even if Ownership differs between a parent type and its descendant, so long as Extensions_Visible is False. Not sure if we want to add such an aspect to Ada.

AARM Reason: We do not allow a nonlimited type to be controlled and also have Ownership True, as the Adjust procedure of the controlled type takes over the semantics of assignment, and would defeat the move/borrow/observe semantics defined below. See H.7.1 for a bounded error relating to the Adjust procedure for a controlled types with any subcomponents with Ownership True.

The Ownership aspect of an anonymous access-to-object type is determined as follows:

- If it is the type of a stand-alone object (including a generic formal **in**-mode object), it is the same as that of the object;
- If it is the type of a formal parameter or result of a callable entity to which the Restriction No_Parameter_Aliasing applies, it is True;
- Otherwise, it is False.

The Ownership aspect of a class-wide type is the same as that of the root type of the class.

An access-to-variable type with Ownership aspect True is called an *owning access type*. Similarly, an object of an owning access type is called an *owning access object*. An access-to-constant type with Ownership aspect True is called an *observing access type*. Similarly, an object of an observing access type is called an *observing access object*. Finally, an object that is a part of an object with Ownership aspect True, or a part of the dereference of an owning or observing access object, is called a *managed object*.

[Redundant: Any composite type with Ownership aspect True is a by-reference type (see 6.2).]

[Ed. note: This additional rule for 6.2, where by-reference types are defined, is given below.]

AARM Reason: Forcing by-reference parameter passing for such composite types simplifies the rules significantly, and avoids the possibility of dangling references upon exception propagation. Later we disallow private types with Ownership aspect True to have a full type that is a by-copy type, so this effectively means that all non-access types with Ownership aspect True are necessarily by-reference types.

AARM Ramification: An observing access object is never an exclusive owner of its designated object; while it exists it may be dereferenced to produce a constant view, which can be passed around, read in parallel, etc.

A name denoting an object can either be a *static name* or a *dynamic name.* The following are *static names*:

- a name that statically denotes an object (see 4.9);
- a selected component with prefix being a static name;
- a dereference (implicit or explicit), with prefix being a static name denoting an object of an access type with Ownership aspect True;

- a name that denotes an object renaming declaration, if the *object*_name denoting the renamed entity is a static name.

Any other name that denotes an object, other than an aggregate or the result of a function call (or part thereof), is a *dynamic name*. Names that denote an aggregate or the result of a function call, or part thereof, have their own set of rules.

AARM Reason: We considered specifying that indexed components having static index expressions were also static names, but chose to leave them out, since they can be aliased with names that have dynamic index expressions. So currently a static name and a dynamic name are never "peer" aliases, and it is trivial for the compiler to determine whether two static names denote the same object. On the other hand, determining whether two dynamic names denote the same or overlapping objects might require a run-time check on the values of array indices, or on the values of pointers of a non owning/observing type.

A static or dynamic name has a *root object* defined as follows:

- if the name has a prefix that statically denotes an object, it is the object statically denoted by that prefix;
- otherwise, the name has a prefix that denotes an object renaming declaration, in which case the root object is the renamed object.

AARM Reason: A renaming can be an implicit borrowing if the renamed object is part of a dereference of an owning access object, so we want to go "through" the renaming, rather than re-dereference the owning access object (which has been implicitly "borrowed" in any case).

Two names with the same root object *statically overlap* when one is a static name and the other denotes the same object, or has a static prefix that denotes the same object.

AARM Reason: We will need this term later, and this seems like the best place to introduce it. Note that static names/prefixes never involve indexing or slicing, so “overlap” is easy to define. They denote the same “whole” object, or one denotes a component or an object “owned” by the other.

A static name that denotes a managed object can be in one of the following *ownership states*:

- *unrestricted* -- the name may be used as defined elsewhere in this standard, and if it denotes an owning access object, a dereference of the name provides a variable view;
- *observed* -- the name provides a constant view, and if it denotes an owning access object, it may be used as the prefix of a dereference, but the dereference is similarly a constant view;
- *borrowed* -- the name is not usable when in this state; the name has temporarily relinquished ownership to a *borrower*;

If a static name is in the observed or borrowed state, then any other static name with the same root object that denotes the same object is in the same ownership state.

AARM Reason: This is a desirable property. We don't want the exact form of a static name to affect its state.

A dynamic name that denotes a managed object can also be in the observed or borrowed state, but in addition it can be in a *dynamic ownership state* where run-time checks may be necessary to determine its (run-time) ownership state. If a name (static or dynamic) that denotes a managed object has a prefix that is in the observed or borrowed state, then the name is similarly in the observed or borrowed state (respectively). If a dynamic name that denotes a managed object is not in the observed or borrowed state because of this rule, then it is in the dynamic ownership state.

> AARM Ramification: In this case we know that any static prefix must be in the unrestricted state rather than observed or borrowed. Run-time checks may be required when using dynamic names that have a dynamic ownership state.

Certain operations are considered to *observe, borrow, or move* the value of a managed object, which might affect whether a name that denotes the object is considered to be in the unrestricted, observed, borrowed, or dynamic ownership state after the operation.

For names that denote managed objects:

- The default ownership state of such a name is determined as follows:
    - If it denotes a constant, other than an **in** parameter of an owning access type, its default state is *observed*;
        > AARM Reason: Constant-ness is transitive for components that are owning access objects, so access-to-constant to the root means that nothing "below" that pointer can be written.
    - If it is a dynamic name that denotes a variable, its default ownership state is *dynamic*;
    - Otherwise, its default ownership state is *unrestricted*.
- The following operations *observe* a name that denotes a managed object:
    - An assignment operation where the target is an access-to-constant object of an anonymous type, and the name denotes an owning access object as the source of the assignment, if the assignment is part of the initialization of a stand-alone object or the parameter association for a formal parameter or generic formal object of mode **in**;
        > AARM Reason: This supports traversing a data structure in read-only fashion, while preserving the handle on the root of the structure.
    - An assignment operation that is used to initialize a constant object (including a generic formal object of mode **in**) of a named type with Ownership aspect True but that is not an owning access type, where the name denotes the source of the assignment.
        > AARM Reason: We handle the case of assignment to named owning access types as a special case below under borrowing. Managed objects that do not have Ownership aspect True can be freely copied, since they necessarily have no nested owning access objects.
    - A call where the name denotes an actual parameter, and the formal parameter is

of mode **in** and composite or aliased;

AARM Reason: Unless the parameter is an owning access object, we treat this as observing. If it is an owning access object, and the formal is access-to-variable, then it is considered borrowing (see below). If the formal is neither composite nor aliased, the parameter is guaranteed to be passed by copy and no long-lived "observer" of the managed object is being created.

At the point where a name denoting a managed object is observed, the state of the name becomes *observed*, and remains so until the end of the scope of the observer. While a name that denotes a managed object is in the observed state it provides a constant view.

AARM Reason: We want the managed object to still designate the same object(s) until all the observers go away, so we disallow updating the object.

At the point where a static name that denotes a managed object is observed, every static name that denotes the same managed object is observed, and every name with that static name as a prefix, is similarly observed.

AARM Ramification: This applies recursively down the tree of managed objects, meaning that observing a managed object effectively observes all of the objects "owned" by that managed object. Dynamic objects rely on a different rule that will check (at run-time if necessary) that there are no objects that "own" this object that are in the borrowed state.

- The following operations *borrow* a name that denotes a managed object:
  - An assignment operation that is used to initialize an owning access object, where the (borrowed) name denotes the source of the assignment and the target is a stand-alone variable of an anonymous access-to-variable type, or is a constant (including an **in** parameter or a generic formal object) of a (named or anonymous) access-to-variable type;

    AARM Reason: This supports traversing a data structure with the ability to modify something, without losing a handle on the root of the structure. Unlike subcomponents of a constant, IN parameters and stand-alone constants of an owning access type still provide read/write access. This is to accommodate existing practice in the use of constants of an access type to still be used to update the designated object. Note that as soon as the object becomes a component of a larger constant, we treat everything "owned" by a composite constant as itself being constant. Note that we don't consider this a complete transfer of ownership (i.e. a "move") because we don't want the sole owner to be a constant, since we can't set it to null if we move the value out of the object.
  - A call (or instantiation) where the name denotes an actual parameter that is a managed object other than an owning access object, and the formal parameter is of mode **out** or **in out** (or the generic formal object is of mode **in out**).

AARM Reason: We consider this to be "borrowing" of the actual parameter name denoting the managed object, while the formal parameter name comes into use as an unrestricted name for the object. We don't want the actual parameter name used for any other purpose during the execution of the called subprogram or generic instance, since the formal parameter name provides full read/write access to the object.

- Object renaming where the name is the *object*_name denoting the renamed object, when the renamed object is not in the observed or borrowed state.

AARM Reason: We are effectively borrowing the object, because we want to be sure the designated object is not deallocated while the renaming exists. If it has already been borrowed, such a rename would not be permitted. If it has already been observed, then the renaming provides a constant view, but that does not change the ownership state of the object -- it remains "observed."

At the point where a name denoting a managed object is borrowed, the state of the name becomes *borrowed*, and remains so until the end of the scope of the borrower. While a name that denotes a managed object is in the borrowed state it provides a view that allows neither reading nor updating.

AARM Reason: We want the name to still designate the same object(s) until the borrower goes away, so we disallow updating via a borrowed name. We also disallow moving, borrowing, observing, and dereference (see below in legality rules). So effectively a name that is borrowed is completely "dead" until the borrowing ends.

At the point where a static name that denotes a managed object is borrowed, every static name that denotes the same managed object is borrowed, and every name with that static name as a prefix, is similarly borrowed.

AARM Ramification: This applies recursively down the tree of managed objects, meaning that borrowing a managed object effectively borrows all of the objects "owned" by that managed object. Dynamic objects rely on a different rule that will check (at run-time if necessary) that there are no objects that "own" this object that are in the observed or borrowed state.

- The following operations are considered *move* operations:
  - An assignment operation, where the target is a variable or return object (see 6.5) of a named type with Ownership aspect True, including an OUT or IN OUT formal parameter of an owning access type.

AARM Ramification: This includes both the copy in and the copy back of such a formal parameter from its actual parameter, and an assignment to/from such a formal inside the subprogram.

  - An assignment operation where the target is part of an aggregate, having a named type with Ownership aspect True.

AARM Reason: These operations are considered "moves" because they

result in a transfer of ownership, and possibly a deallocation of the object designated by the target beforehand, or a setting to null of the source afterward (see dynamic semantics below). Note that this does not include by-reference parameter passing, which is what is used for all composite/private types with Ownership aspect True.

Legality Rules

If a type has a partial view, the Ownership aspect may be specified explicitly only on the partial view, and if specified True, the full type shall not be an elementary type nor an untagged private or derived type with Ownership aspect False; furthermore, if the Ownership aspect for the full type would be True if not explicitly specified, the Ownership aspect of the partial view shall be True, either by inheritance from an ancestor type or by an explicit specification.

AARM Reason: If the full type might contain an owning component, then it is important that the partial view indicates that. On the other hand, if the partial view has Ownership True, it is OK if the full view is tagged but would not be considered to have Ownership aspect True merely due to its components or parent. We disallow having a full type that is untagged private or derived with Ownership aspect False, because such types might be by-copy types, and we want private types with Ownership aspect True to always be by-reference types, to avoid problems with exceptions propagation leaving dangling references.

The source of an assignment operation to an object of an anonymous access-to-object type with Ownership True (including a parameter) shall neither be an allocator nor a function call.

AARM Reason: This could lead to a storage leak, since we do not finalize and reclaim the object designated by such an access object when its scope ends -- we assume there is still some other access object that designates it. An allocator or function call must first be assigned to an (owning) access object of a named type, and then it can be “walked” using an object of an anonymous type. Note that we want any allocators to be of a named pool-specific type so their storage is allocated from a pool that is being used for “managed” objects, so you would end up having, in any case, to put an explicit type qualification around the allocator to specify the named pool-specific type. So requiring the target to be an access object of a named (pool-specific) type doesn’t seem like a big burden.

The prefix of a name shall not denote denote a function call, or a qualified expression whose operand is a function call, an aggregate, or an allocator, if the prefix is of a type with Ownership aspect True. [Redundant: A function call, aggregate, or allocator of such a type can be assigned to an object of a named type (including a parameter), which can then be used as a prefix for selecting components or dereferencing.]

AARM Reason: Again, this is to avoid storage leaks, from constructs like F(A).X, where it is not clear when we could deallocate the storage of the object designated by F(A). We

could relax this by requiring the compiler to deallocate these objects at the “right” time, but that would require worrying about observes and borrows, etc. Also, allowing names whose prefix is a function call or qualified expression with allocator or aggregate as an operand would complicate other rules, so we make them illegal.

An Access or Unchecked_Access attribute reference shall not be of a type with the Ownership aspect True, unless the prefix denotes a managed object, and the value is directly used to initialize a stand-alone object of an anonymous access type, or an **in** parameter of an anonymous access type; this is considered *observing* the prefix if the anonymous access type is access-to-constant, and *borrowing* otherwise.

AARM Reason: We allow taking ‘Access of aliased “managed” objects to initialize an object of an anonymous access type to support use of pointers in programs where dynamic allocation is not permitted. The same rules for borrowing and observing apply, but no “move” operations are permitted, nor any deallocations.

A declaration that observes or borrows a managed object shall not occur within the private part or private descendant of a package, nor within a package body, if the root of the name of the managed object denotes an object whose scope includes the visible part of the package, unless the accessibility level of the declaration is statically deeper than that of the package.

AARM Reason: We disallow borrowing in "private" an object that is visible, since the compiler needs to "know" whether an object has been observed or borrowed, everywhere within the scope of the object.

In a conversion [Redundant: (explicit or implicit)], the Ownership aspect of the operand type and that of the target type shall be the same. [Redundant: Note that anonymous access types are never convertible to (named) pool-specific access types (see 4.6).]

AARM Ramification: A specific type cannot be converted to a class-wide type if the Ownership aspect of the specific type differs from that of the root type of the class. Similarly, a type cannot inherit a dispatching operation from its parent type if the parent has a different value for its Ownership aspect, since the implicit conversion associated with calling an inherited subprogram would violate this rule.

SPARK Note: We could perhaps relax this in the context of the Extensions_Visible attribute being True.

For an assignment statement where the target is a stand-alone object of an anonymous access-to-object type with Ownership aspect True:

- If the type of the target is an anonymous access-to-variable type (an owning access type), the source shall be an owning access object denoted by a name that is not in the observed state, and whose root object is the target object itself;

  AARM Reason: At its declaration, such an owning access object can be initialized from any owning access object in the unrestricted state. On subsequent assignment statements, such an object can only be set to point to somewhere in the "tree" headed by its prior value. Hence, you can use it to walk

exactly one tree -- you can't jump between different trees using the same SAOAAT.

- If the type of the target is an anonymous access-to-constant type (an observing access type), the source shall be an owning access object denoted by a name that is in the observed state, and whose root object is also in the observed state and not declared at a statically deeper accessibility level than that of the target object.

    AARM Reason: At its declaration, such an observing access object can be initialized from any owning access object in the unrestricted or observed state, but on subsequent assignment statements, it can only be assigned from objects rooted at an observer that lives at least as long as this observer.

For managed objects:

- While the state of a name that denotes a managed object is observed, the name shall not be moved nor borrowed and shall not be used as the target of an assignment;

    AARM Reason: Observed objects and everything “owned” by them are read only.
- While the state of a name that denotes a managed object is borrowed, the name shall not be moved, borrowed, nor observed (directly or indirectly), and shall not be used as a primary, as a prefix, as an actual parameter, nor as the target of an assignment;

    AARM Reason: The object has been borrowed, and should not even be observed other than via the borrower.
- If the source of a "move" is a name that denotes an object with Ownership aspect True, other than a function call, the name shall be a variable [Redundant: that is not in the observed or borrowed state]; furthermore, there shall be no name that statically overlaps this name that is in the observed or borrowed state;

    AARM Reason: After a move, we set the source to be null if it is an object other than an anonymous return object, aggregate, or allocator, and effectively “destroy” everything bearing a name that overlaps the source. Note that the only case where a (non-anonymous) owning access object can be a constant and not be in the observed state, is when it is an **in** parameter. So the first part really only disallows moving the value of an **in** parameter. You *can* borrow the value of an **in** parameter, by passing it further along as an **in** parameter, or assigning it to an object of an anonymous owning access-to-variable type. The second part is more important -- you can’t move something if anything “beneath” it is in the borrowed or observed state.
- At the point of a call, any name that denotes a managed object that might be written (other than via a formal parameter) as part of the invocation of the target callable entity (as restricted by the value of its Global aspect and the restrictions imposed by the No_Parameter_Aliasing restriction -- see below) shall not be in the observed or borrowed state. Similarly, any name that denotes a managed object that might be read (but not written) as part of the call, shall not be in the borrowed state.

    AARM Reason: We want to be able to assume that all relevant writable globals are unrestricted when a subprogram starts executing, and all read-only globals

are not borrowed.

Dynamic Semantics

When the value of a variable of a named owning access type is moved to another object, the value of the owning access variable is set to null. This applies to each subcomponent that is of a named owning access type in a move of a composite object. [Redundant: This includes the case of moving an actual parameter to a formal parameter of a named owning access type that is of mode **out** or **in out**, as well as the copy back.]

AARM Reason: If we didn't set it to null, an exception might allow the old value of the variable to be used after handling the exception, which would be bad if the designated object had been deallocated prior to raising the exception.

When a new value is assigned to a variable of a named owning access type, if the prior value is non-null and does not match the new value, the object designated by the prior value is finalized, and its storage deallocated. If the value being assigned matches the prior value of the target, there is no effect. This applies to each subcomponent having a named owning access type, in an assignment to a composite object.

AARM Reason: When an owning access object of a named type can legally be assigned, that means there are no borrowers or observers in existence, so it is safe to reclaim the storage. Objects of an anonymous type never truly "own" an object, though they can be the unique borrower. Note that we don't want to rely on the general finalization rule (see below) for handling the left-hand side of an assignment, because we have the extra test that the new and old values differ.

Similarly, if a variable of a named owning access type (including a component) is non-null when it is finalized (other than when it is the target of an assignment, which is covered by the above rule), the object designated by the variable is finalized, and its storage is deallocated. This applies as well to formal parameters of mode **out** and **in out** that are of a named owning access type, when the subprogram does not complete normally. [Redundant: This rule applies recursively to subcomponents of the designated object that are of an owning access type, ensuring that all objects designated indirectly by an owning access object are finalized and deallocated when the owning access object is finalized.]

AARM Reason: We treat both the copy-in and the copy-back as a "move" for OUT and IN OUT parameters of a named owning access type, so if a subprogram does not complete normally, no copy-back occurs, and we must perform the finalization action to avoid a storage leak.

AARM Ramification: The finalization that normally precedes an assignment, according to 7.6, is handled differently, as described for the assignment case (the finalization only happens if the value is changing). On the other hand, the finalization that happens to the subcomponents of the “old” designated object prior to it being reclaimed, *is* intended to be covered by this rule, so we get reclamation of the entire tree of objects.

Finally, if a constant of a named type with Ownership aspect True is initialized by a function call, aggregate, or allocator, then when it is finalized, if the constant is not itself part of a return object or aggregate, each part of the constant that is of a named owning access type is finalized as above. [Redundant: This includes **in** parameters initialized by such an expression.]

AARM Reason: We are attempting to prevent storage leakage via constants that effectively "own" an object created at the time of their initialization, if initialized by a function call, aggregate, or allocator.

For managed objects denoted by a dynamic name, checks are used to ensure that no other name that is a *potential alias* is in a conflicting state. Two names are *potential aliases* when:

- both names statically denote the same entity;
- both names are selected components with the same selector and with prefixes that are potential aliases;
- both names are indexed components, with prefixes that are potential aliases, and corresponding indexing expressions that are both static, if any, have the same value;
- both names are slices, with prefixes that are potential aliases, and the corresponding discrete_ranges are overlapping in the case when both are static ranges;
- one name is a slice whose prefix is a potential alias of the other name;
- one is a slice and the other is an indexed component, with prefixes that are potential aliases, and for the case when both the discrete_range and the indexing expression are static, the value of the indexing expression is within the range;
- both names are dereferences, with prefixes that are potential aliases;
- both are aliased views having the same type, and at least one of them is a dereference of a non-owning access type;
- one denotes an object renaming declaration, and the other denotes the same renaming declaration, or is a potential alias with the *object*_name denoting the renamed entity.

Two names N1 and N2 *potentially overlap* if some prefix of N1 is a potential alias of N2, or some prefix of N2 is a potential alias of N1. The prefix and the name that are potential aliases are called the *potentially aliased parts* of the potentially overlapping names.

For a dynamic name D1 that is in a dynamic ownership state, if an operation requires that, were it static, it not be in an observed (or borrowed) state, then for every other dynamic name D2 that is in the observed (or borrowed) state and potentially overlaps with D1, a check is made that the potentially aliased parts of the names do not in fact denote overlapping parts of the same object. If this check fails, Program_Error is raised.

AARM Ramification: This requires keeping track of what potentially aliased (see above) dynamic names are observed (or borrowed), and doing checks that there are none that overlap with this name or any part of the tree rooted at this name. Note that this can be as simple as comparing the addresses of the shared prefixes.

AARM Reason: We don’t just talk about “overlap” of the denoted objects, but instead talk about overlap between the potentially aliased parts of the two names. That is necessary because we are interested in a more general notion of “overlap” between names that treats designated objects as overlapping with the object containing the pointer to them.

[Ed. question: Note that we are relying here (and later) on the meaning of (physical) “overlap” that is already used in various parts of the standard (e.g. A(3/4) and 13.3(73.6/3)). Should we actually define it “officially” somewhere -- perhaps as part of defining attribute function Overlaps_Storage?]

H.7.1 Operations of Owning Access Types

The following attribute is defined for an object X of a named non-limited type T:

X’Copy

The evaluation of X’Copy when T is a type with Ownership False yields an anonymous object initialized by assignment from X; when T is a composite type with Ownership True, X’Copy yields an anonymous object initialized from X (including adjustment of controlled parts) but with all subcomponents X.S that are of a named owning access type (and are not subcomponents of a controlled part) being replaced by X.S’Copy; when T is an owning access type, X’Copy raises Program_Error if the designated type of T is limited, and otherwise it yields null if X is null and if not null, the result of evaluating an allocator of type T with the designated object initialized from X.all’Copy. For the purposes of other language rules, X’Copy is equivalent to a call on a function that “observes” X.

If the evaluation of X’Copy propagates an exception, any object that has been newly allocated and adjusted is finalized, including any owning access subcomponent that has been successfully replaced by its copy.

AARM Ramification: The intent is that no newly allocated storage is lost if an exception is propagated by X’Copy, but that no part of X or X.all itself is finalized or deallocated as a side-effect of such an exception.

The stream-oriented attributes (see 13.13.2) for every subtype S of a named access-to-variable type T with designated type D and Ownership aspect True, have the following semantics:

AARM Ramification: Essentially the access objects "disappear" in the streaming representation, and all you have are the non-access type elements of the "tree," plus some Boolean "existence" flags to indicate whether an access object was null.

S’Output

If the designated type D does not have an available Output attribute, Program_Error is raised. Otherwise, S’Output (Stream, Item), where Item is of type T, invokes Boolean’Write (Stream, False) if Item is null, and otherwise invokes Boolean’Write (Stream, True) followed by D’Output (Stream, Item.all).

S’Write

S’Write (Stream, Item) is equivalent to S’Output (Stream, Item).

S’Input

If the designated type D does not have an available Input attribute, Program_Error is raised. Otherwise, S'Input (Stream) invokes Boolean'Input(Stream). If this returns False, S'Input returns null. If Boolean'Input returns True, an allocator of the form:

new D'(D'Input (Stream))

is evaluated and returned by S'Input. If the evaluation of D'Input propagates an exception, S'Input propagates the exception, and no storage is allocated.

S'Read

S'Read (Stream, Item), where Item is of type T, invokes S'Input (Stream) and assigns (moves) the result to Item. If S'Input propagates an exception, S'Read propagates the exception.

The owning access type T in the above is considered to support external streaming (see 13.13.2), provided its designated type D supports external streaming. For the purposes of this determination, a set of mutually recursive owning access types do not by themselves prevent an enclosing type from supporting external streaming.

Bounded Errors

If a nonlimited controlled type T has a subcomponent of an owning access type, it is a bounded error if the Adjust procedure for the type does not replace all such subcomponents with null or an access value designating a newly allocated object. The possible consequences are that Program_Error is raised immediately after invoking Adjust, or the guarantee of an exclusive single updater might be violated, and a future dereference of the unreplaced access value could lead to erroneous execution.

H.8 The No_Parameter_Aliasing Restriction

[TBD: Talk about aliasing between parallel tasklets, or have a separate No_Data_Races restriction?]

This section describes the No_Parameter_Aliasing restriction, which disallows passing a part of a variable to a subprogram or entry, if it is potentially referenced as a global by the callable entity, unless both access paths are read only. Furthermore, it disallows operating on overlapping parts of a single variable twice within the same expression or simple statement, unless both operations are read only, or one operation is read only, and it occurs *strictly before* the second operation where it might be updated.

AARM Reason: We could allow two updates, if one occurs strictly before the second update, but for now we disallow two updates to the same object within a single expression or simple statement, given the likelihood for error.

An operation that occurs within an expression or simple statement occurs *strictly before* a second such operation only if:

- The first occurs within an expression, and the result of the expression is an operand to the second;
- The first occurs within the left operand of a short-circuit control form, and the second

occurs within the right operand of the same short-circuit control form;

- The first occurs within the condition or *selecting*_expression of a conditional_expression, and the second occurs within a *dependent*_expression of the same conditional_expression; or
- The first operation occurs strictly before some other operation, that in turn occurs strictly before the second.

Static Semantics

The following *restriction*_identifier is language defined:

No_Parameter Aliasing

If the No_Parameter_Aliasing restriction applies to the declaration of a callable entity, then the following rules apply on calls to such an entity; these rules also apply if the restriction applies at the point of a call or other operation:

- On any call to which the restriction applies, a part of a variable shall not be passed as a parameter if a statically overlapping part (see H.7) of the variable can be referenced as a global during the invocation of the called entity (as limited by its (explicit or default) Global aspect -- see 6.1.2), unless both access paths are read only;
- Within a single expression or simple statement, statically overlapping parts of a single variable shall not be passed twice as operands to operations, unless both are read only operands, one is a read-only scalar operand that is not an aliased parameter, or one is a read-only operand to an operation that occurs strictly before the operation where it an updatable operand;

AARM Reason: We allow non-aliased scalar operands because they are certain to be passed by copy; we disallow access-type operands if not used strictly before the second operation, because of the possible side-effect of nulling out an access object as part of passing it as an in-out parameter.

Dynamic Semantics

In a context where the No_Parameter_Aliasing restriction disallows static overlap between two names, and the names potentially overlap (see H.7), a check is made, after evaluating the potentially conflicting operand(s) but before invoking the operation(s), that there is in fact no overlap between the potentially aliased parts of the two names. Program_Error is raised if this check fails.

[end of H.8]

Add after 6.2(7/3):

- a composite type with Ownership aspect True (see H.7);

!discussion

The objective of this pair of features, the Ownership aspect and the No_Parameter_Aliasing restriction, is to allow pointers to heap objects to be used safely and efficiently without fear of dangling references or storage leaks, and to reduce aliasing to the point where programs using pointers and pass-by-reference can nevertheless be fully analyzed, in particular for compile-time detection of data races.

The Ownership aspect is a relatively simple idea, where an object is owned by exactly one pointer at any time. When an object is owned, it can be deallocated when its owner is assigned to be null or point at a different object. We want to allow read-only copies to be used to walk a data structure, so we allow access-to-constant copies so long as they are of a type that goes away before the original object is finalized. We also allow access-to-variable copies which are considered “borrowers.” Most commonly these will be stand-alone objects of an anonymous access-to-object type, though an IN parameter of a named access-to-variable type also acts as a borrower.

Similarly, a long-lived dereference of the root of a data structure (e.g. as part of parameter passing) is considered borrowing and may be used to walk a data structure destructively, while restoring the ownership state of the access object designating the root after the dereference goes away. To change what is the root object itself requires a direct assignment to the owner of the root, which cannot occur while the “borrower” exists.

As noted we have four states for an access object: unrestricted, observed, borrowed, and dynamic. An *unrestricted* access object is the owner, and provides read-write access to the designated object. Assigning a new value to an unrestricted access object causes the prior designated object to be deallocated. An *observed* access object is a shared owner, and provides read-only access to the designated object. An observed access object cannot itself be set to point elsewhere. A *borrowed* access object is not useful until the borrowers go away.

Tracking the state of stand-alone access objects is pretty straightforward. Tracking the state of components is more challenging. We have given the rules in terms of static and dynamic names, and their prefixes. We believe these rules are now sound and sufficiently flexible to permit “normal” kinds of data structure manipulations.

We have provided “deep” copying and streaming attributes, so that arbitrarily complex data structures built using owning access types can be easily copied or streamed, without additional work by the programer.

For the No_Parameter_Aliasing rules, we fall back on run-time checks to prevent aliasing when necessary for "dynamic" names. Hopefully such checks can be eliminated at compile-time in

almost all interesting cases. SPARK would complain if such a check could not be proved at analysis time.

We presume that the Adjust procedure of nonlimited controlled types turn every assignment into a “deep” copy, potentially making use of the ‘Copy attribute on any subcomponents that have Ownership True. Therefore, we do not allow a controlled type to itself have Ownership True, even if it has subcomponents with Ownership True, so there is no notion of move/observe/borrow with respect to controlled types per-se, though objects designating them could still have Ownership True. Effectively they are treated like “elementary” types from the point of view of most of these rules. However, they still have to abide by the No_Parameter_Aliasing rules, which are largely independent of Ownership, except in terms of the definition of “overlap.”

So in some sense, these Owning access types provide a very low overhead “controlled” type. If you still want to write your own Adjust and Finalize procedures, then you can explicitly declare a controlled type. If all you want is automatic storage management for levels of indirection, then just use an access type with Ownership True and the ‘Copy attribute where you want it, and let the compiler take care of all of the necessary grunt work of preventing dangling references and storage leaks (in a provably safe manner).

The work on this topic was started in earnest by an intern at AdaCore, Georges-Axel Jaloyan, mentored by Yannick Moy. Many of these concepts are inspired by the "borrow checker" of the Rust language, and/or the anti-aliasing rules of the ParaSail language and the SPARK language.

Eventually we will extend this to cover checking for data races between potentially concurrent actions.

Safety Properties

The rules that make up the above proposal are intended to ensure the following desired safety properties:

1. For any heap object that was created using an allocator of an owning access type, and not yet deallocated, one of the following is true:
    - There is exactly one name in the *unrestricted* state denoting an owning access object that designates the heap object; or
    - There are one or more names in the *observed* state denoting access objects that designate the heap object.
    - When an operation *observes* or *borrows* using a dynamic name, that name enters the *observed* or *borrowed* state throughout the scope of the observer or borrower; during that period run-time checks are used to check any other dynamic name in the *dynamic ownership state* that is *potentially overlapping* with

the first dynamic name, to see if it does in fact overlap, and if so, to preclude it from all operations, if it overlaps a borrowed name, or from updating operations, if it overlaps an observed name.

There may be any number of names in the *borrowed* state denoting access objects designating the heap object, but these give no access while in this state.

2. When such a heap object is deallocated, there are no access objects designating it.
3. Names in the *borrowed* state provide no access to the designated object.
4. Names in the *observed* state provide read-only access to the designated object, and any name that denotes a subcomponent of the designated object that is of an owning access type, is also in the *observed* state.
5. Only static names in the *unrestricted* state provide read-write access to the designated object, or dynamic names in the *dynamic ownership state* where they have been checked to ensure they do not overlap with a dynamic name in the *observed* or *borrowed* state.

!examples

Here is an example of swapping two elements of a singly linked list.

```
type List;
type List_Ptr is access List with Ownership;

type List is record
   Next : List_Ptr;
   Data : Data_Type;
end record;

procedure Swap_Last_Two (X : in out List_Ptr) is
  --  Swap last two elements of list
  --  (no effect if list has less than two elems)
begin
  if X = null or else X.Next = null then
    -- List does not have two elements
    return;
  else if X.Next.Next = null then
    -- List has exactly two elements
    declare
      Second : List_Ptr := X.Next;
        -- Second cannot be a constant because it
        -- will be nulled after a “move”
    begin
      --  Swap them (via “move”s)
      X.Next := null;  -- not really necessary (already null)
```

```
      Second.Next := X;
      X := Second;
    end;
  else
    --  More than two elements; find second-to-last element
    declare
      Walker : access List := X;
    begin
      while Walker /= null loop
        declare
          Next_Ptr : List_Ptr renames Walker.Next;
        begin
          if Next_Ptr /= null
            and then Next_Ptr.Next /= null
            and then Next_Ptr.Next.Next = null
          then
            --  Found second-to-last element
            declare
              Last : List_Ptr := Next_Ptr.Next
              -- Last cannot be a constant because it
              -- will be nulled after a “move”
            begin
              --  Swap last two
              Next_Ptr.Next := null;  --  not really necessary (already null)
              Last.Next := Next_Ptr;
              Next_Ptr := Last;
              return;  --  All done
            end;
          end if;
        end;
        --  Go to next element
        --  Note that we need to do this *after* the “borrowing”
        --  inherent in the renaming of Walker.Next is complete.
        Walker := Walker.Next;
      end loop;
    end;
  end if;
end Swap_Last_Two;
```

!ASIS

[Not sure. It might be necessary to have a query for the new aspects. - Editor.]

!ACATS test

ACATS B-Tests and C-Tests are needed to check that the new capabilities are supported.

!appendix

[elided]