


Safety Unit No. 1: Theory of Alias Sets

Authors: bqe, (add yourself)
Status: Draft 
Created: 2025-09-26
Docs stored in [Carbon's Shared Drive](#)
[Access instructions](#)


"Safety Unit" is a series of docs with "units of discussion": a doc to help written, async communication. It is free form, for instance Question-Answer based. Prefer just writing into the doc instead of commenting. Anything goes. If something good happens here, we will incorporate the content elsewhere.


Context

An alias set (place set) refers to an abstract set of places. It (over) approximates a set of places used at runtime. For now, these are written like A , B

At a given point in time, we know things about alias sets.

Effects change what we know to be true about alias sets.

See  Carbon Language - Memory safety effects (wip)

and  Carbon Open discussions minutes 2025-04 to present (ongoing) .

Unit of Discussion

What are (meaningful, useful) things that we can express about alias sets?

Out of scope: effects, actual syntax

Questions

Q0. What do we need alias sets for? How do these help temporal safety?

[bqe] Tentative answer: in a strict Carbon program fragment, we want every memory location to be associated with some alias set A , and:

- for every access operation we want the compiler to prove "any place from A is safe to access"

- more generally, we may use them for other kinds of properties on memory check, where we can use the information to check whether an operation is permitted (which involves describing the effects).

We shift away from "lifetime" to "access validity" and use effects to track what alias sets are safe to access and which are not.

We can associate extra information like "immutable" with alias sets, which can be checked against the *effect* of operations.

SPECIFICATION NEEDED: what is this extra information, and how is it associated with alias sets?

Q1. Do alias sets form a lattice?

[bqe] Tentative answer: Yes. If we regard them as sets of places, the following operations make sense:

$\wedge A \leq \wedge B$ // roughly: subset or equal. containment.

$\wedge A = \wedge B$ // equality

// should be logically equivalent to $\wedge A \leq \wedge B$ and $\wedge B \leq \wedge A$

$\wedge A + \wedge B$ // roughly: set union

$\wedge A \& \wedge B$ // roughly: set intersection - is it useful?

$\perp, \{\}$ // bottom, empty set

$\top, \wedge any$ // "all places" - maybe not very useful (*)

Further, we can talk about "least upper bound" (simply union of sets).
In informal discussion, we may want to use place variables $p \in \wedge A$.

We may also want to associate program variables with places $x : \wedge A$

(*) when we allocate (transition from one state to another), this changes meaning.

Q3. What about member access, element access, dereference?

[bqe] Tentative answer: these are "projections" that map an alias set to another.

We can have a rich set of alias set expressions that include user-defined operations (member access `x.foo`, element access `x[i]`).

For instance if all places in A can be associated with a type T that has member `foo`, we have *projection* (mapping)

$^A.foo = \{ q \mid p \in ^A \text{ and } p.foo = q \}$ p, q places

We can also talk about *any* projection

$^A.any = ^A.m$ for all members m , element access $^A[i]$ and dereference $*(^A)$

Alternatively, we can describe what happens with projections.

"if $x : ^A$, then $x.foo : ^B$ "

"if $x : ^A$, then $x[3] : ^B$ "

"if $x : ^A$, then $*x : ^B$ "

Q4. Don't these need to be safe to access?

[bqe] Tentative answer: yes, whenever we make a projection $^A.foo$, we implicitly assume that A is safe to access.

If we know that $^A = \{\}$, then we ~~should consider $^A.foo$ a nonsensical expression.~~
also know that $^A.foo = \{\}$.

Q5. What can we say about alias sets?

[josh] Why we care:

...

var $p: T ^A$ * = ...;

var $q: T ^B$ * = p ;

...

is allowed if B contains the place where $*p$ is stored (which is an element of A , but we may have more specific flow-sensitive knowledge)

[bqe] ``Tentative answer:

$disjoint(^A, ^B) \Leftrightarrow ^A \& ^B = \{\}$ // let's not use \perp

[josh] why we care about disjoint: it means an effect on ^A is known to not apply to places in ^B . For example: a [[write(^A)]] is allowed when we require ^B to be immutable.

$\text{overlap}(\text{^A}, \text{^B}) \Leftrightarrow \text{^A} \& \text{^B} \neq \{\}$ // let's not use \perp

$\text{reachable}(\text{^A}, \text{^B}) \Leftrightarrow \text{overlap}(\text{^A}, \text{^B}) \text{ or } \exists \text{ ^C} \leq \text{^A. any with } \text{overlap}(\text{^C}, \text{^B})$

$\text{separate}(\text{^A}, \text{^B}) \Leftrightarrow \text{not reachable}(\text{^A}, \text{^B}) \text{ and not reachable}(\text{^B}, \text{^A})$

(see Q8 why we may care about separate)

Q6. An effect is something that changes what we know about alias sets.
How express that?

[bqe] tentative answer: any assignment syntax

e.g. with ' (tick) suffix to stand for next version $\text{^A}' = \text{^B}; \text{^B} = \{\}$ right-hand side of assignment cannot mention "next" alias sets $\text{^A}'$

e.g. $\text{assign}(\text{^A}, \text{^B}), \text{clear}(\text{^B})$

Q7. Are alias sets enough to express exclusivity?

[bqe] No. We can only talk about exclusivity relative to other alias sets.

So we can say $\text{separate}(\text{^A}, \text{^B})$ for every ^B in scope.

There may however be other alias sets that are not in scope.

And there may be a ^C from which ^A is reachable.

Q8. When we use effects to clear an alias set ^B , how can we deduce that access to ^A has been invalidated?

[bqe] if we know $\text{reachable}(\text{^A}, \text{^B})$, we can construct an example path that leads to empty alias set.

Q9. How can we express shared ownership with alias sets?

[bqe] There is no inherent notion of ownership or exclusivity in alias sets. The only exception is when we allocate a new object, then temporarily we know that we have exclusive access to that object, because it is not reachable from anywhere, including alias sets that are not in scope.

So something is shared when it is in $\wedge B$, and that $\wedge B$ is reachable from another alias set.

Example (A, B classes):

```
...  
var b: B* = new {msg = "hello"}  
var a: A = {.b = b} // A owns b  
send_to_other_thread(a) // a no longer live  
other_module::foo(b) // b reachable from a, but a no longer live  
  
// in other_module  
func foo(b : B*) [[free( $\wedge b$ )]] { ... /* no idea of A */ }  
...
```

Maybe `send_to_other_thread` should have effect `[[free($\wedge A$)]]` which entails `free($\wedge B$)` so that `b` is not usable.

What if it doesn't? This program should still be rejected, as the situation is one where we cannot guarantee safety.