67

Merging state and references breaks ownership

If we accept that we can modify the language to make HasMut<T> and HasRef<T> non-destructible, and to enforce they are not used after a move, then we might consider to go a step further and do away with these troublesome types.

We might try to instead make the reference types MutRef<T> and Ref<T> not-publicly-destructible but also movable with a destructive move. Then we can eliminate the HasMut and HasRef types, and encode those states by the existence of the reference types.

However, that allows a method to steal ownership from a reference. By constructing a Uniq<T> from a MutRef<T>, ownership is taken without being passed a Uniq<T> explicitly. Thus we actually need the states representing HasMut and HasRef to remain in the original scope of the Uniq<T> they are transitioned from in order to return ownership back to the same scope (though not the same variable).

Conclusion

We attempted to represent ownership and borrowing through the C++ type system, however the language does not lend itself to this. Thus memory safety in C++ would need to be achieved through runtime checks.

Merging state and references breaks ownership

If we accept that we can modify the language to make HasMut<T> and HasRef<T> non-destructible, and to enforce they are not used after a move, then we might consider to go a step further and do away with these troublesome types.

We might try to instead make the reference types MutRef<T> and Ref<T> not-publicly-destructible but also movable with a destructive move. Then we can eliminate the HasMut and HasRef types, and encode those states by the existence of the reference types.

However, that allows a method to steal ownership from a reference. By constructing a Uniq<T> from a MutRef<T>, ownership is taken without being passed a Uniq<T> explicitly. Thus we actually need the states representing HasMut and HasRef to remain in the original scope of the Uniq<T> they are transitioned from in order to return ownership back to the same scope (though not the same variable).

Conclusion

We attempted to represent ownership and borrowing through the C++ type system, however the language does not lend itself to this. Thus memory safety in C++ would need to be achieved through runtime checks.

“However, the language does not lend itself to this. Thus memory safety in C++ would need to be achieved through runtime checks.”