Carbon Language - http://github.com/carbon-language

# Open discussions minutes (2021 Aug-Dec archive)

**PLEASE DO NOT SHARE OUTSIDE CARBON FORUMS**

## 2021-12-23

- Attendees: josh11b, mconst
- Mixins
  - Wrote up issue #1000
  - Expect the data member option to be popular
  - Concern: awkward for a list container class to find the corresponding intrusive mixin member to use
  - May end up putting more API than you'd like into the intrusive mixin member
  - Could possibly use a "pointer to member" type parameter to establish that link once
- Operator overloading
  - Main concerns: implicit conversion issue, cyclic dependencies between impls, symmetry requirements for `CommonType` and the two comparisons
  - How okay is it that the interface you implement is different from the interface you use as a constraint?
    - For comparison, you might implement `A as ComparisonI(B)` and use `T:! ComparisonC(A)` as the constraint with `T == B`.
    - Do we want to consistently make these different?
    - [mconst] If we persistently need to have two interfaces, one the user implements, and another that is computed by some magic from the former, maybe that indicates that this isn't the right extension mechanism
    - issue #998 considers other options
      - specially named members are less typing, but not as flexible where they can be defined, and are less natural for requiring multiple functions to be implemented
    - If using two interfaces becomes a best practice, how convenient vs. painful does it end up?
    - Would it be a useful feature to be able to use different things using the same name based on whether it is being implemented or being used as a constraint?

        - Maybe enough to distinguish between "explicitly implemented by the user" vs. "implemented via this block of automatic implementation code"?
            - Added implementations would be things like implicit conversions or symmetric comparison
            - This would allow you to break the cycle, though a cycle detector would still be needed
            - Would be more natural for the user
            - Would give a place to put in things like "give an error if both `A as Comparison(B)` and `B as Comparison(A)` are implemented"
            - Equivalent to being able to say: `weak impl` instead of `impl`, and be able to express a constraint "is strongly implemented"
                - a default impl is strong
                - a default constraint accepts weak impls
                - but probably don't want people to write their own weak impls of other people's interfaces
                - maybe only the interface author is allowed to write weak impls, and also "is strongly implemented", but that isn't as crucial.
            - Advantages over 2 interfaces:
                - more natural for the user; feels like you are implementing a interface instead of opting into some arbitrary customization point that eventually causes an interface to be implemented
                - fewer mistakes getting the name wrong
                    - can prevent people from implementing the constraint version using a final blanket impl
                    - harder to prevent people from using the implement version as a constraint
                -

## 2021-12-21

- Attendees: josh11b, mconst
- Dynamic scoping
    - Idea: reverse generics in order to define a static logging class
    - Allocators?
        - Games will know that some type will always want to allocate from the "level arena", which it can know about statically
        - RPCs might have an arena per RPC call
            - protos will want to allocate from that arena
            - some allocations in the RPC are *not* from the arena, for example state updates
        - What about standard containers?
            - In C++ they default to a standard global allocator
            - Other case is pre-allocated container with a fixed maximum size

        - calling code provides max size as a template parameter, memory is part of the type's size, container never touches the dynamic allocator
        - C++ standard allocator mechanism didn't make it easy to just use std::vector with a custom allocator
      - Example: string interning, map from hash to pointer into arena with length + string data
      - temporary data structures using standard containers whose lifetime is the same as an RPC
    - Thread-local allocator, to avoid slowdown from accessing thread-local storage
      - ideal would be the compiler automatically hoisting out the thread-local lookup out of code that is allocating a lot
    - Conclusion: allocators are not implicitly dynamically scoped. Sometimes they are explicit parameters (like in RPCs). Trying to make a class have a general pluggable allocator frequently doesn't work as well as making a new class (interned strings and fixed max size containers).
- Carbon: mixins
  - data members best handled by making mixins into data members, then base classes, and finally interfaces/constraints
  - interfaces/constraints and base classes both have ways of saying methods are required/defaulted/provided, but are different and have different defaults
  - Most minimal thing: data members could just be allowed to know their containing type and their offset within it
    - Containing type wouldn't even have to say this is a mixin, this is pretty safe
    - Enough for things like intrusive linked lists
    - Would have to manually forward any functions/methods that you want to be part of your type's API
    - Serializer mixin is going to iterate through all data members, but may need to recurse to find which have members with a Serializer type
      - Awkward if members of Serializer are dependent on the main type
      - For Serializer specifically, probably only the body of the Serialize method is dependent
      - Would be nice if metaprogramming isn't affecting the interfaces of types, for incremental compilation and IDE completion
  - Could also allow base classes to be parameterized by their final derived type
    - Like CRTP in C++, but more reliable
    - Concern is wanting to get rid of multiple inheritance
  - No use cases we could think of for multiple serializer members of a type
  - what about mixin like a base class minus the things that make multiple inheritance scary
    - The mixin could use constraints (including template constraints) on the main type parameter to say that it has certain functions
    - How do you provide a default other than virtual, and we don't want to be using dynamic dispatch in this case

        - Could just have another keyword (`default`?)
        - the mixin is not an object, so you can't have a pointer to it, so there is no situation where you could accidentally call the mixin version when the main type defines an override
        - So `mixin` is a keyword used in place of `base class`
        - members are not allowed to be `virtual` or `abstract`, but they can be `default`
    - In common between data member approach and base class approach
        - mixin declaration includes a main type parameter, which may be generic or templated, and may have constraints (has interface, extends base class, and template constraints like has method or data member)
        - need a way to convert between a pointer to a data member (in the data member case, this would be the pointer to the mixin; in the base class case, this would be a pointer to a member of the mixin) and a pointer to the containing main type
    - Intrusive linked list class has a friend intrusive linked list mixin with the data, but the class can do the magic conversions
        - need some way to address the mixin of a main type using the parameter that makes it unique

## 2021-12-13

- Attendees: josh11b, zygoloid
- Talked about generic plans, with goal of supporting operator overloading
    - principle: open extension points (operator overloading, swap and other ADL extension points from C++) all are through interfaces, see
      Carbon closed function overloading proposal
    - If we don't want `A op B` to be inconsistent with `B op A` ever in the standard library, then would like to only define it once. Example: would like to enforce consistency ordered and equality comparisons (preferable, not necessary)
    - operator overloading: implicit conversion issue, cyclic dependencies, symmetry requirements like `CommonType`, hierarchy of interfaces: the baseline interface, one with implicit conversions, a homogenous one, numeric
        - existing work defining interfaces for operators
    - a way to abstract over implementation; goal is to be able to define an `F` that takes an implementation of `X` and provides an implementation of `Y` and `Z`. Application: for implicit conversion, for A+B giving a definition of B+A, similarly A<B -> B<A.
- Use cases for dynamic scoping
    - Errors
        - [josh11b] In January, made a proposal where the error handler decides whether (& which) errors are expected
    - Other effects
        - async
        - Logging

        - I/O that can be swapped out for tests
        - Language with effect inference: https://effekt-lang.org/
    - Implicit context object or implicit parameters
        - https://odin-lang.org/docs/overview/#implicit-context-system
    - Concern: pervasive sources errors
        - Rust/Swift happy with having potential exits marked, can't be done if errors are too pervasive
    - Overflow? Probably undefined behavior in performance mode, exits program in hardened?
    - Allocation?
        - Some embedded users will want to either forbid allocation or handle allocation errors
        - Allocators are an application of dynamic scoping in multiple languages
    - Problem with using async for everything is the danger of changing your function into a state machine with all state on the heap, terrible for performance.
    - Exciting, but a bit experimental for Carbon
        - choices are: too little support (Rust), bespoke support for each effect (Swift), or exotic general approaches (research languages)
        - Likely set aside for now and come back to, maybe after we go public and we can draw from the wider community including academia
    - Capabilities?
        - Thread-safety annotations (implemented in Clang using its capabilities support) are not perfect, but maybe that is because it is hard in C++ to know when things are destroyed and therefor mutexes are released
        - Capabilities more generally useful for writing secure code (object capabilities avoid the confused deputy problem), running 3rd party code with less trust, dependency injecting for tests

## 2021-12-08

- Attendees: josh11b, mconst, chandlerc, zygoloid
- [josh11] non-wild card impls, but a mismatch between the interface's signature for `Add` and the signature used in the type's definition (but we want to allow that difference since we can perform implicit conversions between the two).

```
class MyInt {
  impl as AddTo(i32) {
    let AddResult:! auto = Self;
    fn Add[me: Self](x: i64) -> Self { ... }
  }
}
```

- [josh11b] wild card impls, implicit conversion to a single function

```
class MyInt {
  impl as AddTo(T:! ImplicitAs(i32)) {
```

```
    let AddResult:! auto = Self;
    fn Add[me: Self](x: i32) -> Self { ... }
  }
}
```

- [zygoloid] One possible principle: ignore the "impl as ..." lines just use the signatures from the class definition
  - downside: but do you get interface defaults?
- [zygoloid] Other choice: whatever is in the interface is authoritative
  - downside: surprising that the signature written isn't the one in the class
  - downside: unreasonable to support internal wildcard impls
- [josh11b] also evolution risks with this decision -- interfaces might be fine initially to be defined internally, but adding a function with a particular signature breaks that.
  - hard error vs. having functions you can't call
- [mconst] Do we want to allow internal wildcard impls but you can't call when there is ambiguity that isn't resolved by the caller
- Two impls with the same function name with the same interface
  - potentially forbidden, but then adding a function to an interface can be a breaking change if it introduces name conflicts with members of another interface
  - this could be a *permanent* breaking change, where we from then on never can internally implement both of two interfaces from different libraries
  - internal impls are a bit like inheritance w.r.t. possible breakage
- Evolution concerns are more about internal impls in general
- [mconst] Simplest alternative is to just use `T` instead of `i32` in the function signature
- [chandlerc] Two interfaces: one has `ImplicitAs`, and calls the other
  - default implementation performs the implicit conversions
  - somehow the interfaces would do this automatically

```
class MyInt {
  fn Add[me: Self](x: i32) -> Self { ... }
  external impl as AddTo(T:! ImplicitAs(i32)) {
    let AddResult:! auto = Self;
    fn Add[me: Self](x: T) -> Self { return me.Add(x as i32); }
  }
}
```

```
// Possibility 1
interface AddToWithImplicit(T: Type) {
  fn Add[me: Self](x: T) -> Self;

  // external?
  impl [U:! ImplicitAs(T)] as AddTo(U) {
    fn Add[me: Self](x: U) -> Self { return me.(AddToWithImplicit(T).Add)(x); }
  }
}
```

```
class MyInt {
  impl as AddToWithImplicit(i32) {
    fn Add[me: Self](x: T) -> Self { ... }
  }
}
```

```
// Possibility 2
interface AddTo(T:! Type) { ... }
interface AddToWithImplicit(T:! Type) { ... }
// Implicitly "for all U"
impl [U:! Type, T:! AddToWithImplicit(U), V:! ImplicitAs(U)] T as AddTo(V) {
  ...
}

class MyInt {
  impl as AddToWithImplicit(i32) { ... }
}
```

- "for all" interpretation of blanket impl, however, that example doesn't need to determine any types from anything other than the `Self` type or interface
  - Don't like how possibility 2 may need to consider an infinite set of possible values for `U` due to blanket impls
- Possibly do a search for implicit conversions specially for operators
  - One concern is interaction with blanket impls
- Invented the manual conversion approach in C++
- Maybe simplify by not allowing implicit conversions for both arguments
  - would not scale since would have to search all impls
- Hack from dynamic languages: have `add_to` and `reverse_add_to`
- In C++, only allows implicit conversions for non-templated overloads, uses ADL to limit where to look
- Put all the implicit conversion logic in the standard library
  - Do we want to allow types to be able to opt-out of automatic implicit conversion?
  - There is C++ code that turns off implicit conversions
    - but C++ has a lot more implicit conversions
    - Blocking an implicit conversion in one API, such as implicit conversion from string literal to a string being blocked for SQL queries
      - have a more specialized version that is faster since it does less checking
    - Writing pointers to a text buffer, special case for `char*`
    - Thing your doing is semantically meaningful, but is a code smell, is lint
  - Could this be a lint feature: don't ever block implicit conversions. Instead just declare that if the source type was X, print an error.
    - Concern that it would not be detected in the generic case

        - Maybe magic template powers? Just like getting the file & line number of the caller in asserts?
- Question: search for impl reachable via implicit conversion either done in the language or in the standard library
    - Chandler&MConst would like there to be a way to get the same behavior for non-operators for user's types; a reason to use a library solution
    - either way: users who want to override an operator just implement an interface once with a non-wildcard implementation, no discrepancy allowed between type and interface
- MConst: evolution risk for allowing type's definition doesn't match interface
- zygoloid: if we do the implicit conversions in the library, how do you present that to users? given this impl with a hole in it, how do you find the right impl? our current syntax doesn't make it clear where that would fit

```
interface AddToWithImplicit(T: Type) {
  fn Add[me: Self](x: T) -> Self;

  impl [U:! ImplicitAs(T)] as AddTo(U) {
    fn Add[me: Self](x: U) -> Self { return me.(AddToWithImplicit(T).Add)(x); }
  }
  impl [U:! ImplicitAs(Self)] U as AddTo(T) {
    fn Add[me: U](x: T) -> Self { ... }
  }
}
```

- zygoloid: Is this going to interfere with defining a wildcard impl for 'AddToWithImplicit'
- zygoloid: Need to be clear on the exact algorithm we use for finding the `AddTo` impls
- mconst: maybe this will end up needing to be imperative rules for how to delegate / find the relevant impls
- Remove mismatching signatures and internal wildcard impls from the proposal, will need to tackle this problem in a future proposals
- Like removing mismatching signatures
    - Concern about calling the same function with the same types but from a template vs. a generic getting different results
    - Other surprising edge cases

```
class MyInt {
  impl as AddToWithImplicit(i32) { ... }
}
```

- Approach 1:
    - Look for impl MyInt as AddTo(i8)
    - Look for impl MyInt as AddToWithImplicit(?)
    - could find i32, but may have some ambiguity if there are other choices
    - concern with doing a lookup for AddToWithImplicit(?) for coherence reasons
- Approach 2:

```
class MyInt {
  // make this more of a template-y mixin -- immediately expanded, no wild cards,
```

```
not looked up
  impl as AddToWithImplicit(i32) { ... }
  // as if
  impl [U:! ImplicitAs(i32)] as AddTo(U) {
    fn Add[me: Self](x: U) -> Self { return me.(AddToWithImplicit(T).Add)(x); }
  }
  impl [U:! ImplicitAs(Self)] U as AddTo(i32) {
    fn Add[me: U](x: T) -> Self { ... }
  }
}
```

    - Does this support wildcard impl of `AddToWithImplicit`?
    - No: this isn't really implementation of an interface, it is more like a templatey mixin
    - Just a short-cut for writing a bunch of parameterized impls that is tedious & error-prone

## 2021-12-06

- Attendees: josh11b, jonmeow, chandlerc, wolffg
- Go-public discussions
- DynPtr(MyInterface) questions
    - Question: Is there some way to write down an interface such that it will generate an error if it isn't object-safe? Is this required for `DynPtr`? Advantage is that object-safe becomes part of the contract, so you don't accidentally add a method to the interface and breaks users. Don't want users to add a non-object-safe method with a default impl just to prevent users from depending on it being object-safe.
- Do we want a principle around metaprogramming where it is maximally powerful but minimally used?
    - Goal is to get good errors, rather than keep the language small by implementing a lot of features using metaprogramming than in the language itself.
    - Minimally used is the more important component: We would be willing to sacrifice metaprogramming power to improve diagnostics, fast compile, etc.

## 2021-11-30

- Attendees: geoffromer, josh11b
- [geoffromer]: Thoughts on the type of a class name, when used as an expression?
    - Tentative agreement that `typeof(ClassName)` is a plausible spelling for that type
- [josh11b]: issue #578 is a possible application, but alternatives exist

## 2021-11-29

- Attendees: chandlerc, josh11b, zygoloid, wolffg, mconst, jonmeow
- Outstanding PRs
  - #624 coherence
  - #931: impl access
- Issue #710 on default comparison for struct types / data classes
- `final let`, impl inheritance, and named impls
  - More comments
- migrating templates from C++ without bringing all of C++'s semantics & syntax
  - can make some things the same, such as static vs. non-static member access
  - idea: create an interface with a C++ templated impl with all the dependent operations used by a templated function
  - big concern: references
  - another concern: lifetime rules for temporaries subexpressions
    - rewrite to something more different can address this by rewriting to a function taking a callback
  - would we consider only doing interop but not automatic migration for templated code?
    - can do better by looking at all instantiations in the code base and see if the template is ever interpreted in different ways. probably would handle enough cases?
    - probably could handle temporary lifetime concerns as long as the rules between C++ and Carbon are similar enough and we can translate each individual statement into an individual statement
    - references: C++ often allows punning between pass-by-value and references, but not in templates and in Carbon we have to make a choice
    - concern: operators not found by ADL
- `final let` again
  - discussing the tight coupling between the impls where there are `final let` like things
  - edge cases two impls with different final associated types and some intersection
  - Carbon: specialization example
  - impl inheritance adds complexity, but might be a nice feature on its own
  - Undecided between final let vs. final impl
  - Need more examples
  - Not even sure if final should be default or not
  - [chandlerc] Worried about complexity; would like to favor simpler rules
    - A reason to start with `final impl` and see how it goes, instead of trying to figure out the edge cases with `final let`
    - Another option: Maybe don't restate `final let` in a specializing impl, even that forbids partial overlap cases
- Again: migrating templates from C++ without bringing all of C++'s semantics & syntax
  - Expect common case is that we don't want to export a templated API
    - can see users, can manually instantiate

- Another case: template is part of API, and not going to migrate all users to Carbon
  - look at a bunch, maybe not all, of instantiations
  - if consistent, can use that consistency for Carbon translation
  - if funny business, keep as a C++ template, migrate implementation details C++ -> Carbon; conservative
- Won't know how common funny business is until we try making a tool
- Difficulty levels:
  - 1. Computer can automatically translate since there is a consistent interpretation
  - 2. Human can see intended translation
    - May need to do idiom recognition
  - 3. Template happens to support different things, but in practice only used in one consistent way
  - 4. Actually does fundamentally incompatible different things
    - [chandlerc] Expected to be rare, could potentially leave that in C++ forever
    - Need to factor into Carbon design, reduce likelihood that translation will introduce different semantics
- Could possibly include a compile-time check that the machine translation is using the same interpretation as the original C++
- Found a company that does tooling for language migrations, have done some few million line codebase migrations
  - E.g. weird custom language -> Java
  - Long tail problems would be solved by matchers for idioms common in a particular codebase
  - Was not a material scaling limitation for them
- Concern: C++ punning between pass by copy vs. pass by reference because they have the same syntax at the caller
  - Hope that the difference generally won't matter
  - C++ behavior commonly is broken
  - Falls down in the case where some instantiations where there is a copy, some where there is no copy, and the code observes the difference; e.g. sees a mutation when there is no copy
  - Concern: not migrating all at once - can we generate Carbon code that calls a function that takes a const reference in some cases and passes by value in others
    - Can possibly make these the same in Carbon
    - Less common but harder problem is const-qualified methods
  - Want Carbon frozen values ("immutable borrows" or "pass by immutable value") will be allowed to passed to const reference parameters seamlessly
  - Edge case: non-copyable values that get mutated, e.g. a large object where the identity matters, passed by const reference and want to observe mutation
    - Common case the type is large, could use pointers

        - Plan is to translate both const reference and pass-by-copy to Carbon frozen values
            - Could also use `var` parameter to force a copy
        - Could also do static analysis
            - How is this template used? Is it sometimes used with copies?
        - Mutable references and pointers both migrate to pointers
        - Try to minimize the amount of level 4 problems
        - Know if we are passing to a Carbon or C++ function, may more eagerly copy when calling C++
        - Frozen passing is mostly to keep values in registers to make a fast calling convention, not about Fortran-style optimizations. Still might be able to do those optimizations, but would be willing to give them up if it hurts interop.
        - Takeaway: For interop purposes generally, at call sites and expressions, want the same sort of punning in Carbon as in C++ to make templates more compatible
            - Not going to do this for class members -- all reference members will become pointers in Carbon
        - How do we support perfect forwarding of a forwarding reference?
            - Look at examples more carefully, but maybe works out
            - Or more invasively changing to lambdas
        - Allow moving from a value consistently with ~, even if it doesn't call the destructor when it is a frozen value
            - Still need to be able to reason about when destructors are run for types where we care; not with types where you don't care about copy vs. move
- Back to Issue #710:
    - Decided that the order of fields in a struct matters
        - but common operations can efficiently deal with the different orders
    - We support converting assignment
        - between types with a set of fields with the same names and convertible types
    - We support heterogenous comparison
        - fields with the same name and comparable types are comparable, both for equality and less than
        - different orders are allowed for equality comparisons, not less than comparisons
        - always does the comparison in the field order of the left operand
        - if you really care how comparisons are done, use a nominal class
    - Explicit cast for passing different field order to a function
        - no implicit conversion
    - Do data classes get these comparisons?
        - Maybe a data class is an adapter of a struct? Allows you to control whether you extend and keep the implementations from the struct
            - But syntactically annoying since it would split up the implementation

            - Would allow you to `as` into the underlying type
        - Generally think that data classes will want all the convenience features of structs:
            - just want a name and some extra methods
        - can later have different variants of data classes by implementing a different interface than `DataClass`
    - Do we want to allow implicit conversions between structs with the same field order and implicit conversions between the types?
        - Argument for: making having two integers returned from a function and passed to another work like a single integer
        - Hesitation from MConst and zygoloid
        - To be viable, don't want as many implicit conversions as C++
        - Non-simple aggregates should be given a name
        - Same argument for other aggregates like arrays.
            - Concern: Array(i8, 1000) -> Array(i32, 1000)
            - Maybe a typo leading to a big copy which wouldn't occur if both Array(i32, 1000)?
        - [chandlerc] Would prefer to distinguish between aggregates vs. not instead distinguishing between things with named fields or other criteria
        - Probably won't pass returned pair to a function without destructuring first
        - Conclusion: Destructuring is what allows you to do the conversion, there is no implicit conversion of aggregates

## 2021-11-15

- Attendees: jonmeow, josh11b, zygoloid
- [zygoloid] Three things want for non-type parameters of a type
    - Want code monomorphized on the value, such as array bounds, may need different code generated for different values
        - [josh11b] Previously said would use template parameters, not generics for this, so some array bounds could be rejected
    - Types that are distinguished by a phantom parameter that isn't going to be used for anything other than causing the types to be different, so we can't e.g. mix up a trusted string with an untrusted string
        - [josh11b] Possibly could answer type equality of these types using `observe` statements, just like other type equality questions
    - To parameterize a type on a runtime value, for example to implement the flyweight pattern. (Possibly the second thing is a special case of this.)
- [zygoloid] Differences between templates and generics
    - Value known when type checking vs. unknown
    - Impl selection vs. validation that some impl applies
        - In a template, can use the values of associated types and other entities from the actual impl selected
    - Whether you use the interface of the type vs. the constraints
    - [josh11b] Can use `if` in a function signature, but only using template values and build constants

    - [josh11b] Template instantiation can fail on use after the definition is first compiled/checked
- [zygoloid] Maybe we only care about templates in the first case, and runtime parameters in the second and third cases? So no need for non-type generics?
- [josh11b] What about a graphics library with a point/vector interface that supports 2, 3, and 4 dims? Should we support an associated constant with `N`?
    - Concern is that associated constants will not have their value known at type checking time, will need to be treated generically
- Rust calls these "const generics", long-awaited feature: https://without.boats/blog/shipping-const-generics/
    - Has "No complex expressions based on generic types or consts" rule similar to what we were discussing for Carbon
    - Getting the size of a generic but sized type is an interesting use case
    - "use cases like cryptographic hash traits which allow each implementation to specify a different length for the hash they output"
    - To get type equality, would need to reason about how expressions are derived from generic parameters
- Compile-time BigInts aren't expensive and are total for + and *, but not /
- Risk of failure from an array bound being to big is in some sense equal to the monomorphization failure from making a type too big by having large members
- Does i32 addition overflow also fit in the category of things that could be allowed but trigger monomorphization errors?
- Philosophically how much do we want to mark things that could fail at monomorphization time with the keyword `template`
    - Don't want "type-too-big-for-implementation" to be marked
    - Do want to include things that can't be definition checked, like division by 0 or a halting problem, as generics
    - Shouldn't promise to definition check expression equivalence, even though it is possible to do, because of the high-polynomial time; instead mark with `template` and accept a late error rather then do an expensive proof that it is always safe
- Allow equality of integer values only if you specify a constraint, like `TypeId` constraint for types allowing `==`
    - Would potentially allow us to avoid monomorphization
    - Maybe only allow `==` in constraints
- [zygoloid] Generically could have single-step identities between integer expressions, and then use `observe` statements to combine them:
    - observe Array((N + M) + 1, T) ==
    - Array(N + (M + 1), T) ==
    - Array(N + (1 + M), T) ==
    - Array((N + 1) + M, T)
    - Can maybe get pretty far with just a few identities
- [chandlerc] Treat generic integers just like types, can address the second use case, but not the first
    - start there, and then see how painful arrays are
    - assume we can't solve all primitive array problems, like testing for negative bounds

    - important that non-type parameters aren't a wart, completely unuseful
- What is going to be blocking?
    - templates are going to be needed for standard library
    - operators:
        - symmetry
        - a way to get an answer for the easy cases in generics, where we can't get an answer for the hard case
- Is it acceptable to have something at the interface level or built-in to the compiler for the equal type case needed for CommonType?
    - Needs to be away to express a special case for equal types beyond just the CommonType case, but it maybe doesn't need to handle other conditions
    - We will encounter this again, don't want to hard-code this in the compiler
- Another example:
    - Does Deref on T* always give T, or can it be specialized?
    - Motivates "this impl can't be specialized further and you can rely on it when type checking"
    - Same mechanism should handle deref T* and CommonType of T and T
- Concern is if we privilege impls in the interface definition, doesn't solve the problem for shared pointer types
- More important to use the associated type of the blanket implementation than other aspects of the implementation. We could safely specialize the implementation of methods, etc.
- [zygoloid] Perhaps for type structure purposes could consider an impl defined inside the class as an exact match for `Self` even if it is parameterized, so can't specialize outside class further.
    - Could opt out by using external out-of-line
    - Possible solution for pointers and shared pointers, but not common type
- Perhaps some rules that impls are not specializable if they are inside the definition of the `Self` type or interface
    - Can always see the ones in the interface, and blanket impls that don't see the type are never prioritized above ones that mention the type
    - Interface1 has a preferred blanket impl for anything T implementing Interface2; type U(V) has a preferred implementation for Interface1; concern is then there is a definition for Interface2 for U(String)
        - Rule: don't allow preferred impls to have overlap in type structure.

## 2021-11-11

- Attendees: jonmeow, josh11b, zygoloid, cjdb, mconst
- [cjdb] Semantic requirements for C++
    - Spectrum: from structural, to nominal, to tested, to formally verified
    - Could take the Python approach where the compiler ignores annotations but other tools use them, moves expensive static analysis off the critical build path
    - Standard C++ concepts do have semantics specific by the C++ spec, but no way to define semantics for user-defined concepts

    - [josh11b] Carbon will probably follow formal verification, not lead
    - [zygoloid] Need for migration from C++ means needing to support incremental migration
    - [josh11b] Improvements to safety and formal verification in Carbon will likely be a large number of incremental steps
- Continuing Monday's discussion
    - [mconst] been thinking about some examples

```
class Set(T:! Hashable) {
    impl Container {
        let Element:! Type = T;
        let Iterator:! Iterable where .Element == T = SetIterator(T);
        fn Begin[me: Self]() -> Iterator { ... }
        ...
    }
    let Iterator:! Iterable where .Element == T = ...;
    fn GetRandomElement[me: Self]() -> T { ... }
}
class Potato {
  fn Hash...;
}
external impl Potato as Hashable {
  fn Hash...;
}

var h: Set(Potato) = ...;
h.GetRandomElement().Hash();
```

    - What type does `h.GetRandomElement()` have?
        - First said `Potato as Hashable` but have changed it to be `Potato`
        - Want `Set(Potato).Iterator` and `Set(Potato).Begin()` to give you a `SetIterator(Potato)` not a `SetIterator(Potato as Hashable)`
    - [zygoloid] One position is that facet types are just a convenience to avoid having to repeat qualification for accessing names from an interface, and it is fine if the particular facet is lost if you go through a function call in a generic context
        - In a non-generic context, have constant types
    - [josh11b] Most important is that user of `Set(Potato)` always gets types as if `T == Potato` instead of an erased type
    - [mconst] Eventually: what happens if `T` is declared as a template instead of a generic in the same `Set` class.
        - Maybe have `Hash` conflicts
        - Do the return types involve `Potato`, `Potato as Hashable`, or `Potato & Hashable`
        - Question in terms of name lookup: Under which circumstances, if any, do we look up names in the type-of-type in addition to the type? If we are doing double name lookup, is it symmetric difference (union-conflicts) or name lookup in the type followed by conflict checking (type-conflicts)?
        - [mconst] less interested in type-conflicts than was on Monday

        - Downside of union-conflicts is if the modified type escapes, it is more visible. But we'll just fix the escapes instead.
    - In the generic case:
        - One easy answer: we don't do any double name lookup, T in the body of the generic is an archetype, and archetypes always internally implement all of their interfaces. Outside the generic, use the type unchanged, no coercion to a facet type.
        - [zygoloid] Equivalent to saying: inside the generic we do type-of-type name lookup
        - Eliminating coercion to a facet type when passing to a generic function
        - [chandlerc] When calling a template from a generic, may want to reject code that doesn't cast to an adapter
    - We had liked M intersect Z, but it has a problem in the template case
        - Plain option M would do double name lookup on template types, in order to help evolution from template -> generic
        - Z always does double name lookup
        - M intersect Z does too, concern that leaks out to caller via return type
        - [zygoloid] One possible rule: only do double name lookup (union-conflicts) when dependent on a template parameter
            - [mconst] agree that this is a solution; more convenient, harder to describe, mostly only helps in the case where
        - [mconst] Another possible rule: never do double name lookup
            - in generics lookup would be in either the archetype or the type-of-type
            - in templates, would behave just like substitution. Manually monomorphization would preserve semantics.
            - in practice this means you would have to qualify accesses to interface members
            - what about the bridge-to-generics story?
                - start with a template have to use qualified names
                - constrained templates still would have to use qualified names
                - generics would not have to qualify
            - concern: a template might not use qualified names
                - if a template wants to call a method from an interface, have to qualify to be reliable, in all of the models under consideration
        - [zygoloid] Simplest option: always qualify if you don't have a constant type, always qualify to access an external impl even if the type is constant
            - makes generic code more verbose
            - but makes the rules uniform across regular code, templates, and generics
            - simple model for templates: means constant propagation, no more no less

[break]

- [chandlerc] Want generic facilities to also be available in non-generic code.

- [mconst] Can we write these in non-generic code:

```
let Foo:! Hashable = Potato;
let template Bar:! Hashable = Potato;
```

- [chandlerc] Should imagine how we get generic behavior when in a non-generic and examine how that works.
- Suppose we're in a function with a deduced return type, and we return an expression of type `Foo`, what is its return type?
- [zygoloid] The "Simplest option" above has the property that generics don't have additional facilities, but at a severe ergonomic cost.
- [chandlerc] Preference: get all the things that we are comfortable having outside generics, make available in generics. Also in templates: people want to write code that's a template but is definition-checked. Eg, code that is mostly generic but needs to be a template because (eg) it calls a template. Still want to type-check everything else.
- What if the API change (from eg `T:! Hashable`) is very localized – your use of that `T is Hashable`, but other people who just use that value, stored elsewhere, don't get the Hashable effect.
- [mconst] Intuitively, only want double name lookup in scope of `T`. Don't yet have a precise formulation of this.
- [chandlerc] Even in a template, if I say `T is Hashable`, I want name lookup to go to `Hashable` (and not `T`).
- [mconst] Problem: can no longer add constraints gradually. Sharp transition from unconstrained -> constrained that immediately changes lookup, so must specify all constraints.
- [chandlerc] Can separate constraints from changes in interface to allow constraints to be added gradually. Eg, different spelling.
- Constraint does not imply you want a different interface, but there may be common patterns and correlation.
    - Syntactically separate constraints from change of interface / change of archetype. Former is for constraining generics / templates, latter is for ergonomics. Change of archetype would imply a constraint but not vice versa.
    - Perhaps `where` does not change archetype.
    - Perhaps scope-based rule. Eg, in associated type, constraint affects the interface only within that interface or impl, not in users outside the scope of the associated type.
    - Type introduced by some type name with constraints, which specify the API. All names which map back to that type introduction get that API.

```
let PV:! Vegetable = Potato;
var s: Set(PV) = ...;
s.GetRandomElement().Hash();
```

        - Expectation: `GetRandomElement()` returns `PV` here.
            - Maybe: perform a symbolic evaluation that stops at `:!` bindings that are still in scope.
        - `Set(PV)` and `Set(Potato)` are in some sense not the same: their `GetRandomElement`s have different return types

        - Perhaps they are the same but have different mappings from (eg) `T` back to local types.
    - Perhaps: `PV` introduces a new opaque  type (an archetype) where all we know is that it implements `Vegetable`, so `Set(PV)` and `Set(Potato)` are different types. But we know we can cast between `Potato` and `PV`, and between `Set(PV)` and `Set(Potato)`.
    - `PV` has the archetype property and the constant property
    - If this were an actual generic:

```
fn F[PV:! Vegetable](...) {
  var s: Set(PV) = ...;
  s.GetRandomElement().Hash();
}
```

      then `Set(PV)` cannot be compatible with `Set(Potato)` since there is no connection between `PV` and `Potato`, you could instantiate the generic with a different `Vegetable`.
    - But if the two types are known to be related the conversion could be OK:

```
fn F[PV:! Vegetable](...) {
  let  PT:! Type = PV;
  // Set(PV) and Set(PT) are compatible
}
```

      - [chandlerc] Would be fine with requiring an additional `where .Self == PV` in the type of `PT` if you want to perform this conversion
- What do other languages do in this space?
    - In Rust, lookup looks in all traits in scope that are implemented, regardless of whether we're in a generic.
        - No way to restrict the lookup to only certain traits other than qualification or writing a generic that's only used with one type argument.
        - Swift is similar, in both cases you are always looking everywhere you could think to look. No difference between external and internal impls, all impls are internal even if declared out-of-line.
    - In Haskell, no member names. First look up independent of types, and then check that the constraints are satisfied.
    - In C#, interfaces are derived from and never implemented externally or by anyone other than the class author. Generics can constrain which interfaces are implemented or more generally which base classes are derived from.
    - Why are we different?
        - Because of separating internal and external impls and trying to give definitive control to the type author of the API exposed by the type.
- What do people like?
    - [chandlerc] Like having `let` type declarations create archetypes, but they don't change type identity at all
        - 3 places: function body, associated type in an impl, member type in a (generic parameterized) class, have an `=`
            - maybe have an implicit constraint that you can convert between the two sides of the `=`

        - associated types in interface declarations are like generic function parameters, don't have an `=`
        - How do we get the return type as the caller expects, when the return type involves an associated type, type parameter, etc.
    - [zygoloid] symbolically evaluate the type (of an expression etc.), resulting in an expression in terms of type parameters (names of archetypes). The declared types of those type parameters determine their interfaces. If the result doesn't involve type parameters (eg, we evaluate all the way to Potato) then we get the interface of that non-generic type instead of an archetype.
        - works except in the let in the function body case, type check as if it were a generic
        - [mconst] another way we could phrase that, not in terms of symbolic evaluation: when type checking the body of a generic, the parameters get instantiated as archetypes, which we think of as a normal concrete type, and then do normal type checking
        - Issue is you want to look though them in an impl but not in a function body
            - you are in the scope of it in the function body
        - [chandlerc] Like the archetype model, but would say that the archetype is always scoped. When you leave the scope you always revert back to the type identity and nothing more
    - [chandlerc] This example tries to construct a case where it tries to return the archetype formed in the body of the function, but the caller still gets the caller's type

```
fn F[T:! Hashable](x: T) -> auto {
  let U:! Hashable = T;
  return x as U;
}
fn G[T:! Hashable](x: T) -> T {
  return x;
}
var p1: Potato;
// Should look in Potato for Hash.
G(p1).Hash():
// This should be rejected ... somehow ...
F(p1).Hash();
```

        - Maybe have to reject `F`, on the basis that it is returning a type that won't be in scope in the caller
        - Maybe also reject `F` because it does return type deduction
    - [mconst] Consider returning an associated type

```
interface Hashable {
  let HashCodeType:! Integral;
  fn Hash[me: Self]() -> HashCodeType
}
fn F[T:! Hashable](x: T) -> T.HashCodeType {
```

```
  return x.Hash();
}
class PotatoHashCode {
  fn Mash[me: Self]();
}
impl Potato as Hashable {
  let HashCodeType:! Integral = PotatoHashCode;
  fn Hash...;
}

var p2: Potato;
// Return type of F(p1) is PotatoHashCode
F(p2).Mash();

let PH:! Hashable = Potato;
var p3: PH = p2;  // names on `p3` are the names of `Hashable`
// error:
F(p3).Mash();
// Assuming `Integral` has an `Add` function, what happens here?
// PH involves an archetype that is in scope, so this succeeds at
// looking up `Add` in `Integral`. [chandlerc] likes this
F(p3).Add(42);

// TO MOVE BELOW
let template PT:! Hashable = Potato;
var p4: PT = p2;  // What are the names of `p4`?
// Potato, Union of Potato & Hashable - Conflicts, or Hashable, Hashable over
Potato
// [mconst&zygoloid pref] Potato [chandlerc pref] Hashable
// Concern with the sym-diff is complication
// Hashable over Potato is more palatable to zygoloid while trying to accommodate
//   Chandler's desire
// Benefit of sym-diff allows transition from template -> generic without ever
silently changing behavior, not true of Hashable over Potato since when you add the
constraint can silently switch what is called
// mconst: provides the most convenient interface, but hard to explain when this
feature activates; particularly when we look at the type-of-type of an associated
type
// zygoloid: I think it'll be surprising w symmetric difference that h.Hash is
ambiguous, despite the Hashable constraint. I think the dev will think they already
qualified it by writing `: Hashable`
p4.PotatoMethod();              // [Potato] yes [Hashable] no [sym-diff] yes
p4.Conflict();                  // [Potato] yes (Potato) [Hashable] yes (Hashable)
[sym-diff] no
p4.HashableMethod();    // [Potato] no [Hashable] yes [sym-diff] yes
// Allowed? [zygoloid pref] yes [chandlerc pref] no [sym-diff] yes
```

```
F(p4).Mash();
// Allowed? [zygoloid pref] no [chandlerc pref] yes [sym-diff] yes
F(p4).Add(42);
// Allowed? [zygoloid pref] yes (in Potato) [chandlerc pref] yes (in Hashable)
[sym-diff] no?
F(p4).Overlap();

fn MyT[template T:! ???](x: T);
// ??? [zygoloid pref] yes [chandlerc pref] yes [sym-diff] yes?
MyT(F(p4));
```

        - [chandlerc] Scope interpretation is that we lose the fact that `HashCodeType` is an archetype outside the `Hashable` interface and `F` generic function
        - [zygoloid] In the model we're thinking of the model, there is no such thing as `HashCode as Integral`; if it isn't in terms of a placeholder == the name of the archetype, or we are not in the scope of the archetype, there is only one type.
        - [mconst] This model is equivalent to the earlier model where we never do double name lookup; name lookup is only done on the actual type. When type checking generic code, we introduce archetypes, which are concrete types that internally implement all their interfaces.
        - [zygoloid] There is *a* way to manually monomorphize a generic, even if it isn't the most obvious thing you might think to write.
    - [mconst] We have rejected the simplest model because it is too verbose
    - [mconst] Now let's talk about templates: two models still being considered differ only in treatment of templates
        - Question: do we want to require that people need to qualify calls in a template? This is purely a question of name lookup
        - Examples showing the difference between the two template models:

Move template example here

        - [zygoloid] Point is that generics are evaluated symbolically, but templates are substituted early, so they are gone before we consider symbolic evaluation. This gives the `Potato` answer.
        - [mconst] easy to describe, works like C++ templates, different from generics but in an expected way
        - [zygoloid] Type of template, constrained, only limits which types can be assigned to the template parameter, really only used in function declarations, mostly meaningless in a function body
        - Don't want a semantic cliff when adding the first constraint to a template
        - [chandlerc] How much of the benefit of generics do we want to provide when using constrained templates? Want to provide as much as possible for the case where the constrained template is expected to persist and is not just a stepping stone to transitioning to generics. Example case is bridge code to C++ templates.

        - Weird case is external interfaces, and external interfaces are going to be rare in the migration case
    - Agreed on a model as facet types, constraints need not be types-of-types, they just induce constraints and define the archetype used
    -

## 2021-11-08

- Attendees: josh11b, jonmeow, chandlerc, zygoloid, mconst
- Agreed: Treat constant and non-constant types differently
- Option Z: name lookup in both type and type-of-type
    - Concern: when `x` and `y` have type `i32`, `x + y` having a different API than `x` or `y`. This would come from the type-of-type changing to `AddableWith(i32)`.
- Option J: name lookup only in type (when type constant) or archetype (when type non-constant)
    - No one advocating for this option at the moment, since we prefer to call templates from generics with the original type rather than facet type
- Option M: J with no facets
- Can still use adapters instead of facets when you explicitly want to project onto the interface of a type-of-type

```
adapter Facet(C:! Constraint, T:! C) for T {
  extends C;
}
// Maybe, though could be fine with just explicit `as`:
external impl [C:! Constraint, T:! C] T as ImplicitAs(Facet(C, T)) { ... }
external impl [C:! Constraint, T:! C] Facet(C, T) as ImplicitAs(T) { ... }
```

- Explaining M: generic function

```
fn F[T:! Serializable](x: T) {
  // Typechecked using T = Archetype(Serializable, "T")
  x.Serialize();
  // resolves to x.(Serializable.Serialize)();
}
F(2 as i32);
```

- `T` implements `C` implies `T` is a subtype of `Archetype(C)`
- Problem with "Agreed": there is no way to specialize on constant vs. non-constant distinction
    - No way to define an `impl` for `i32 as ___` that is specialized for `i32` instead of applying to all types
    - Hard to write a blanket impl that applies to all facets of a type
- Explaining M: associated type

```
impl i32 as AddableWith(i32) {
  let AddResult:! Type = i32;
```

```
  fn Add(...) -> AddResult;
}
var x: i32 = 1;
var y: i32 = 2;
x+y // result has type i32 (technically "as Type")
```

- Explaining M: associated type with constraint

```
interface HashValue {
  fn Combine...;
}
external impl i32 as HashValue { ... }
interface Hashable {
  let HashType:! HashValue;
  fn Hash[me: Self]() -> HashType;
}
external impl i32 as Hashable {
  let HashType:! HashValue = i32;
  fn Hash...;
}
var p: i32 = 1;
var h: auto = p.(Hashable.Hash)();
h.Combine();  // Allowed in Z but not M
h + 1;  // Allowed for both M, Z
```

- Use case: calling several functions from an external interface, works in Z

```
interface DrawingContext {
  fn DrawRectangle...;
  fn DrawCircle...;
  fn SetPen...;
  fn SetFill...;
  ...
}
external impl Window as DrawingContext { ... }
fn Render(w: Window as DrawingContext) {
  w.SetPen(...);
  w.SetFill(...);
  w.DrawRectangle(...);
  ...
}
```

- Possibility with M: The types of all types could be `Type`.
- Could use subtyping to represent "satisfying a constraint"

```
[W:! Type ◁ DrawingContext](w: W)
(w: some DrawingContext) // static dispatch
(w: DrawingContext*) // dynamic dispatch
```

    - Swift regrets not marking when forming an object trait, Rust is switching to using `dyn`
- `W:! DrawingContext` could be sugar for `W:! Type where W is DrawingContext`, so type is still `Type`, just have a restriction
- in Z:

```
// These replace the type-of-type from caller
fn F[template T:! Type](x: T)
// or:
fn G[template T:! Hashable](x: T)

// vs. keeping type-of-type from caller
fn H[template T:! auto](x: T)
```

- in M: If the type is known at type-checking, use the type's API, otherwise use the type `Archetype(Constraint)`

```
fn F[T:! Hashable](x: T) {
  // is resolved to `x.(Hashable.Hash)()`
  x.Hash();
}
var i: i32 = 1;
// and i32 implements Hashable
// calls i.(Hashable.Hash)()
F(i);
```

- `T` is a subtype of `Archetype(Constraint)` if `T is Constraint`
- in Z: `Window as DrawContext`
- in M: `Window as DrawContext` is probably a mistake

```
impl i32 as AddableWith(i32) {
  // Type is not load bearing here
  let AddResult:! Type = i32;
}

fn F(x: SomeVariant) {
  match (x) {
    case Widget:
    case Gadget:
  }
}
```

- have types A, B, and C each of which implements interfaces `I1` or `I2` and `J1` or `J2`. `I2` extends `I1`, `J2` extends `J1`. This example comes up in ASTs.

```
ABC = Variant(A, B, C)
fn F(x: ABC) {
  match (x) {
    case [T:! I2] x: T => { ... }
  }
```

```
  if x.Type() is I2 {
    // Doesn't require stating the type, which might be dynamic
    x as Archetype(I2)
    x as Facet(I2, x.Type())
  } else {
    // x.Type() is I1
    x as Archetype(I1)
  }
}
```

- Another example where we want to create generics inside a function

```
fn StrCat[T:! GenArray(ToString)](args: T...) {
  var result: String = "";
  // type `U` is different every iteration
  for [U:! ToString] x: U in args {
    result += x.ConvertToString();
  }
  // Possible alternative approach:
  args.Call(lambda [U:! ToString] x: U {
    result += x.ConvertToString();
  });
  // C++ way
  (result += args.ConvertToString, ...);
  return result;
}
```

- Motivation for M:

```
interface AddableWith(RHS:! Type) {
  let Result:! AddableWith(RHS);
  // Vs.
  let Result:! Type where .Self is AddableWith(RHS);
  fn Add[me: Self](y: RHS) -> Result;
}
external impl i32 as AddableWith(i32) {
  // In M, `Type` is superfluous
  let Result:! AddableWith(i32) = i32;
}
var x: i32 = 3;
// In Z: type of `x + x` is `i32 as AddableWith(i32)`
(x + x).Add(x)

interface HashValue {
  fn Combine[me: Self](w: Self) -> Self;
}
impl i32 as HashValue { ... }
```

```
interface Hashable {
  let T:! HashValue;
  fn Hash[me: Self]() -> T;
}
impl i32 as Hashable {
  let T:! HashValue = i32;
  ...
}
var x: i32 = 1;
x.Hash().Combine(x);
```

- Concern about the difference between a generic vs. non-generic context; perhaps the important distinction is internal vs. external; if we have two `Add` methods, one external from `AddWith`, and one internal that means something different, don't want to suddenly get a value from `x + y` where `Add` means something else than it does elsewhere in the function
- Hashable is a motivating example, also containers
- Name lookup and compiler error messages are motivating
- In Z, could make the type of `i32` be `Type`.
- Question: do we agree about the intersection of M and Z, the examples where they agree
- Question: what are the situations where doing double-name lookup in option M would be different from plain option M?
- Pass a `String` to a generic function and get back `String as Hashable`, in variant M would get an error if you use a name that is in conflict between `String` and `Hashable`
- Can already make return type be regular `String` from functions as long as the return type was not from an associated type
- Option Z but the only places you can write a type-of-type are in the parameters of a generic
- Still need type-of-type as a constraint on associated types in interfaces
- But can write them in a way such that associated types only implement interfaces externally, which would make it more similar to M
- In the option M model, the caller's type is preserved when calling a generic, but it does use a different type when type checking
- Chandler requests some examples to show the differences between the options, including container examples
- intersection is:
    - "option Z with no facet types"; option Z but we never coerce anything to a facet type
    - option M except that we perform double name lookup, not just in template contexts, but in all contexts (generic contexts double lookup == single lookup)
- Double name lookup means: we lookup names in the type, that is the name we use, however if the type-of-type also has the name and it conflicts, then we fail. This gives us the migration property we need for migrating from templates to generics, but avoids injecting names from external implementations.
- `i32 as Hashable` would not be legal because M and Z give different interpretations
- Could make double name lookup could also allow names only in the type-of-type to resolve

- Could also consider transitions from generics to templates
- Z: Like having a unified name lookup. Z always looks in the value, type, type of type, etc. and requires only one result
- M: Finds type minus conflicts from type-of-type rule equally simple
- Largely converged! Main question is what double-name-lookup rule we want.
- Intersection is restrictive, but seems to cover the range of expressiveness that we want, but we should check.
- Should do an example of a function whose return type is an associated type.
- When you are writing an associated type, it always has type-of-type `Type`, but it could have constraints that it implements some interfaces.
- The only places where you get a type-of-type other than `Type` are templates and generics.
- [chandlerc] Want to be able to write code that behaves like a generic inside of a non-generic, for example in a match statement.
- Missing from option M, so not in the intersection.
- Principle: generic should not add expressivity beyond their genericness

## 2021-11-02

- Attendees: josh11b, zygoloid
- Have two rules for prioritization, have shown they are different, both seem viable, neither so far has a reason to prefer it
- Expect that we will have some rule for avoiding cyclic impl dependencies, result will be informal rules that users follow to avoid introducing cycles, those informal rules will be necessarily be pretty conservative to deal with the uncertainty about what is happening in other libraries; but don't know what those informal rules will look like yet
- Should avoid cycles with CommonType by introducing the symmetric interface that is implemented only by blanket rules
- Could we forbid users from implementing `AutoCommonTypeWith`, by saying the name is not exposed/public (private in the API file) and so types can't mention the name, but still see which types satisfy it for purposes of the `CommonType` constraint
    - AutoCommonTypeWith is an implementation detail of the CommonType constraint, how CommonType determines if its constraint is met
- Error messages?
    - Motivation for `static_assert` is being able to recognize situations and generate specific error messages that can be clearer
    - Downside is that it introduces the possibility of failure during impl selection
    - Maybe we can give another way of specifying the error message when a constraint is not satisfied; "Constraint, say why you weren't satisfied"
    - Something we can do with constraints and not interfaces, due to the open extensibility of interfaces
- Can we do better than just a block of text to report when a constraint is not satisfied?
    - Would like to have different errors depending on what went wrong, but how to write that?

- `impls_in_priority_order` can give you a "not"; actually since one impl is a subset of the other, specialization gives it to you

```
class TrueType {}
class FalseType {}
interface NotFooImpl {
  let T:! Type;
}
impl [T:! Foo] T as NotFooImpl {
  let T:! Type = FalseType;
}
impl [T:! Type] T as NotFooImpl {
  let T:! Type = TrueType;
}
constraint NotFoo {
  extend NotFooImpl where .T == TrueType;
}
```

- This definition does not give you a law of excluded middle

```
interface X {}
impl [T:! Foo] T as X {}
impl [T:! NotFoo] T as X {}
fn F[T:! Type]() {
  // will not type check because we have no "law of excluded middle"
  ... T as X ...
}
```

- Reason is because in theory someone could further customize `NotFooImpl`
- Can also implement "`and`" and "`or`", so can develop arbitrary boolean expressions on which interfaces are implemented for a type
- How ergonomic is this functionality available to constraints?

```
constraint CommonType(U:! Type) =
  if Self as CommonTypeWith(U) and not U as CommonTypeWith(Self)
  then ...
  else if Self as CommonTypeWith(U) and U as CommonTypeWith(Self)
  then if (Self as CommonTypeWith(U).Result != U as CommonTypeWith(Self).Result)
     then error "CommonTypeWith defined in both directions between {U} and {Self}"
  else ...
```

- Need to have consistent members any time we don't have an error, for `CommonType`, want a `Result`
- May also want to know that certain types implement certain interfaces
- Three things:
  - Criteria whether a type implements constraint
  - Error message if type doesn't implement constraint, preferably saying what went wrong
  - What a function can assume about types that satisfy the constraint

- Nice to have: ability for a type to implement a constraint directly in cases where that makes sense (currently this is determined by whether there are aliases for all names that must be implemented)
- Can we compose constraints to avoid combinatorial explosion when trying to state constraints?

```
constraint CommonTypeHelper(U:! Type) {
  let Result:! auto =
    if U is ManualCommonTypeWith(Self)
    then U.(ManualCommonTypeWith(Self).Result)
    else if Self is ManualCommonTypeWith(U)
    then Self.(ManualCommonTypeWith(U).Result)
    else if U is ImplicitAs(Self)
    then Self
    else if Self is ImplicitAs(U)
    then U
    else error("...");
}

constraint CommonType(U:! Type) {
  let Result:! auto =
    if U is not CommonTypeHelper(Self)
    then error("...")
    else if Self is not CommonTypeHelper(U)
    then error("...")
    else if (...Result == ...Result)
    then ...Result
    else error("...");
}
```

- Unfortunately, the composition here loses the context that would let us deliver a clear error message. How bad is it if we do this directly without a helper?

```
constraint CommonType(U:! Type) {
  // alias Result = ... ?
  let Result:! Type =
    if T == U
    then T
    else if U is CommonTypeWith(Self)
    then
      if Self is CommonTypeWith(U)
      then
        // U.Result would be longer unless we do an `if let` above
        if U.Result == Self.Result
        then U.Result
        else error("inconsistent CommonTypeWith")
      else U.Result
    else if Self is CommonTypeWith(U)
```

```
    then Self.Result
    else if Self is ImplicitAs(U)
    then
      if U is ImplicitAs(Self)
      then error("ImplicitAs in both direction")
      else U
    else if U is ImplicitAs(Self)
    then Self
    else error("nothing is true");
}
```

## 2021-11-01

- Attendees: josh11b, zygoloid, jonmeow, chandlerc
- What constitutes a cycle that Carbon will reject?
    - `impl [T:! Printable] Optional(T) as Printable` -> not a cycle
    - `impl [T:! Type, U:! ComparableTo(T)] U as ComparableTo(Optional(T))` -> not a cycle
    - Definition of circular: During query "`A as B`?", circular if the answer depends on whether `A as B` is implemented.
    - The other kind of failure is infinite search. Example: `A is B` if `Optional(A) is B` if `Optional(Optional(A)) is B` if ...
        - Could be the result of a single impl `impl [A:! Type where Optional(.Self) is B] A as B { ... }`, or a chain of impls
    - Concerning example:

```
interface Foo {
  let X:! Foo;
}
impl [A:! Foo where .X is B] A as B { ... }
impl [T:! Type] T as Foo {
  let X:! Foo = T*;
}
// or maybe:
class Bar(T:! Type) {
  impl as Foo {
    let X:! auto = Bar(Optional(T));
  }
}
```

    - Circularity is a property of a query
    - A set of impls may or may not have cyclic queries.
    - If we can determine an impl even though there is a cycle, should we accept or reject?
    - Similarly for infinite -- maybe there is another, more specific impl that determines `A as B`
    - Should we reject cycles in the absence of a query?

    - Could eliminate infinite towers if we require every step to get no more complicated.
    - Cycles could be spread out across libraries with no dependencies between them
        - Query: `Vector(Pair(A, B)) as Map(C, D)`?
        - `impl [T:! E] Vector(Pair(A, T)) as Map(?, ?)` in `LibA`, and so on
    - Options (from shifting the error left to right)
        - We SOMEHOW have a restriction that prevents cycles only checking each impl definition
        - Combining impls could give you an immediate error
        - A cyclic query could give you a late error (and maybe an infinite search is terminated by a recursion limit)
        - A cyclic query leads to an arbitrary but deterministic choice
    - All of these options are bad, in different ways. Generally speaking, concern is problems created by a library that are only detected by its users.
    - Question (re: first option): Can we restrict blanket impls so they are acyclic by virtue of using the partial ordering of dependencies?
        - Seems tricky, since we want to support writing impls that relate interfaces from independent libraries for a local type
- Talked about explicit prioritization of impls by including them in an ordered block in a file
    - Resolves questions when type structure is the same
    - Can be generalized to apply to mixing type structures: implicitly defines intersection impls. Example:
        - Library1: has `(?, ?, A, ?)` prioritized over `(B, ?, ?, ?)`
        - Library2: has `(?, ?, ?, C)` prioritized over `(?, D, ?, ?)`
        - What impl is selected for the query `(B, D, A, C)`?
        - Prioritization means:
            - Library1 implicitly defines `(B, ?, A, ?)` delegating definition to `A`
            - Library1 implicitly defines `(?, D, ?, C)` delegating definition to `C`
        - Winning type structure is `(B, ?, A, ?)`, which delegates to definition `A`
        - Possible way to rephrase this rule:
            - Pick the impl pattern most favored for the query, then pick the definition of the highest priority matching impl in the block
            - NOTE: LATER FOUND A COUNTER-EXAMPLE, discussed in Discord
            - Counter-example:
            - Library3 has `(A, ?, ?, D)`, `(?, B, ?, D)`
            - Library4 has `(A, ?, C, ?)`
            - `(A, ?, C, ?)` from Library4 is most favored type structure of those explicitly listed for query `(A, B, C, D)`
            - However, Library3 implicitly defines intersection `(A, B, ?, D)` which is favored more
    - We like prioritization for impls. Prioritization is only required among those with the same type structure.
    - Can we use the same explicit prioritization rules for overloads?
    - This doesn't immediately generalize to function overloading because we want to be able to favor exact matches over implicit conversions.

    - Possible resolution: each function overload implicitly generates 2^N impls with different type structures, and then match using the most specific type structure
    - Match statements also have to consider implicit conversions like function overloading, unlike impl selection
- Principle: Order can only matter for items that are collocated and contiguous ("together in a group")
    - Not okay: order of things in file scope mattering for e.g. prioritization, particularly if other declarations can be intermixed
    - More important, though, to allow users to put public members before private members in a class
    - Would be okay to have a public API forwarding to data members that are together in layout order. You might have the rule: you can't have both public & private members, but if you have private members you can still have public properties forwarding to them.
    -
- Talked about https://csis.pace.edu/~bergin/slides/Maclennan.html , but didn't feel like we agreed with the principles strongly enough to adopt them
    - Concern about inconsistency helping user sentiment in some cases
    - But Dart gets good sentiment from seeming familiar
        - `println!` looks like shouting in Rust, seems disconcerting
    - To be familiar to C/C++ programmers, use same pointer syntax
    - Need to be different enough that it won't be confusing whether it is C/C++ or Carbon
    - Carbon has been heavily prioritizing "Manifest interface: All interfaces should be apparent (manifest) in the syntax."

## 2021-10-29

- Attendees: josh11b, mconst
- See Carbon: blanket impl contradiction

## 2021-10-25

- Attendees: chandlerc, josh11b, zygoloid
- [zygoloid] Concern that implied constraints may be dependent on a template parameter.

```
fn F[template T:! Type, U:! Type](x: T, y: U) -> T.G(U) {
  return x.H(y);
}

class RealT {
  class G(U:! Hashable) { ... }
  fn H[U:! Type, me: Self](y: U) -> G(U);
}
```

    - Claim: neither of the `U:! Type` declarations need to be replaced by `U:! Hashable`, that is implied (as in "implied constraint").

    - However, the following `U:! Type` is not okay, and does need to be replaced by `U:! Hashable`

```
// Not Okay
fn Call[U:! Type](x: RealT, y: U) { F(x, y); }
```

    - When type-checking the template `F` (and its body), do not have any inferred constraint on `U`. After instantiating `F` w/ `T = RealT`, the resulting generic does have implied `U:! Hashable`.
- Coherent specialization rules
    - Assumption/requirement: no cyclic dependencies between libraries
    - Type structure of an impl
        - Given an impl declaration, find the type structure by deleting implicit parameters and replacing type parameters by a `?` (also replace `T*` by `Ptr(T)`)
        - The type structure of this declaration:

```
impl [T:! ..., U:! ...] Foo(T, i32) as Bar(String, U) { ... }
is:
impl Foo(?, i32) as Bar(String, ?)
```

    - Type structure → orphan rule
        - Orphan rule: given a specific type and specific interface, impl that can match can only be in libraries that must have been imported to name that type or interface
        - Only the implementing interface and types (self type and type parameters) in the type structure are relevant here; an interface mentioned in a constraint is not sufficient since it need not be imported
    - Type structure → resolves overlap
        - At most one library can define any impls with a given type structure
        - Consequence of both the orphan rule and no cyclic dependencies
        - Corollary: two impls defined in different libraries must have different type structures
        - Given a specific concrete type, say `Foo(bool, i32)`, and an interface, say `Bar(String, f32)`, want an overlap rule that selects the type structure of the impl to select
        - 1) Write down the *unique* type structures of all *matching* impls, for example:

```
impl Foo(?, i32) as Bar(String, ?)
impl Foo(?, ?) as Bar(String, f32)
```

        - 2) Our rule is to pick the type structure with a non-`?` at the first difference
            - Here we see a difference between `Foo(?, i32)` and `Foo(?, ?)`, so we select the one with `Foo(?, i32)`, ignoring the fact that it has another `?` later in its type structure
        - Corresponds to a depth-first traversal of the type tree to identify the first difference. Could also do another order, such as breadth-first, but this order is both simple and reflects some experience from the Rust community that the `Self` type is particularly important to prioritize.

- Same type structure --> 1) same ~~library~~ API file, 2) intersection requirement (or "semi-lattice" requirement)
  - Once we've determined the type structure of the matching impl, we need to pick between the matching impls with that type structure
  - These impls will necessarily be defined in the same library together
  - Compiler will check that for every pair of overlapping impls in a given library, either:
    - one matches a strict subset of the other
    - there exists another impl exactly matching their intersection
  - Result is there is a unique most specific impl with a type structure, using the partial ordering of impls by containment
  - An impl mentioning a private type may be private to the file defining the private type, but otherwise impls must be declared publicly in an API file
- Consequences
  - Can be assured an impl exists for all specific types that match a general blanket impl, but it might be some specialization that is not visible to a generic function. Big improvement in usability since it greatly reduces the need to list "and parameterized type/interface is implemented" requirements
    - CAVEAT: Except template instantiation can fail. Does not change the assumption that it won't when an impl is selected.
  - Specialization makes the dynamic strategy harder
- Example: `interface Addable(T:! Type)`

```
impl [M:! BigInt, U:! ImplicitAs(Int(M))] Int(M) as Addable(U);
impl [N:! BigInt, T:! ImplicitAs(Int(N))] T as Addable(Int(N));
```

  - These rules on their own are fine, the first is used on their overlap. Can also add:

```
impl [N:! BigInt] Int(N) as Addable(Int(N));
```

  - This last rule would be given priority over the other two rules
- Rust considering a "Child trumps parent" rule "This rule is particularly simple and is usually what you want when such overlap arises."
  - Problem: simple application of the rule is not transitive:
    - (A, ?, ?) parent
    - (?, B, ?)
    - (?, ?, C) child
    - A < B < C < A
  - Fix: "Child trumps parent on their intersection" rule
    - 2: (A, ?, ?) parent
    - 3: (?, B, ?)
    - 4: (?, ?, C) child
    - 1: (A, ?, C) delegates to 4
  - Without some child trumps parent rule: If I define a new type, then all impl lookup for interfaces implemented by that type as Self will consider impl from my library first, at the time I define it until some other library adds an impl of that type as Self.

            - However, adding the "child trumps parent on their intersection" rule removes this property.
        - Does `U:! FacetOf(T)` imply `U:! ImplicitAs(T)` because there is a requirement in `FacetOf` on `ImplicitAs` or because there is a blanket impl? *either* is acceptable.
- Want to prevent `FacetOf(T)` from being implied by users. Should we allow users to declare impls as "final"? No, that is a problem for our "resolving ambiguous overlap by declaring an impl on the intersection" plan. Instead just say that `FacetOf(T)` is a compiler-provided constraint, not something users can impl.
- Would like to allow forward declaration of impls inside classes.
    - Need a syntax for defining the impl for the class out of line
    - Inside the class, we need the assignments to associated types (plus anything else that would affect type checking), but the methods are effectively already forward declared in the interface.
- Do we want blanket impls to have (optional?) names?
    - Benefits:
        - would allow delegation between impls
        - would allow out-of-line definition
        - would allow you to call a method from a specific impl to implement an impl in terms of another without just delegating
    - Can we frame blanket impls as providing a facet type that has the needed impl?
    - [zygoloid] One way to name impls:

```
facet RightAddableInt(M:! Bigint, U:! ImplicitAs(Int(M)))
  = impl Int(M) as Addable(U) {
  fn Add[me: Int(M)](v: Int(M));
}
fn RightAddableInt(...)Add[me: Int(M)](v: Int(M)) {
   ...
}
```

        - [chandlerc] The parameters to `RightAddableInt` look like explicit parameters, not things that are deduced
        - Could we leave out the declaration of `Add` before its definition? Concern is that it accepts a different type than is defined by the `Addable` interface, but a type that can be implicitly converted to
    - [chandlerc] Another idea, which has desirable properties but is awkwardly very verbose/redundant:

```
facet RightAddableInt(M:! Bigint, U:! ImplicitAs(Int(M))) {
  impl Addable(U);
}
impl [M:! Bigint, U:! ImplicitAs(Int(M))]
  Int(M) as Addable(U) = RightAddableInt(M, U);
```

        - Approximately equivalent to existing adapter solution
    - [zygoloid] Impls do already have a name, though it is long and awkward to say, that would allow you to perform some of these tasks:

```
// Define a method out of line
fn [M:! Bigint, U:! ImplicitAs(Int(M))] (Int(M) as Addable(U)).Add(...)

// Delegate a method to another impl
impl [N:! BigInt] Int(N) as Addable(Int(N)) {
  fn Add(...) = [M:! Bigint, U:! ImplicitAs(Int(M))] (Int(M) as Addable(U)).Add;
}
```

    - Contrast with the earlier named approach:

```
// Delegate a method to another impl
impl [N:! BigInt] Int(N) as Addable(Int(N)) {
  fn Add(...) = RightAddableInt(N, Int(N)).Add;
}
```

    - The named approach is shorter, and more specific; since to convey the same info we'd need to write:

```
// Delegate a method to another impl
impl [N:! BigInt] Int(N) as Addable(Int(N)) {
  fn Add(...) = [M:! Bigint, U:! ImplicitAs(Int(M))]
    (Int(M) as Addable(U) where M == N and U == Int(N)).Add;
}
```

    - Maybe we allow you to re-open an impl in the same file? Should have a distinct syntax, and we're already using the keyword `impl`
    - We could also do a thing like the proposed use of `@` in patterns, where you write:

```
impl [...] RightAddableInt(M, U) @ (Int(M) as Addable(U))
```

    - The `RightAddableInt(M, U) @` part would be optional
- Question: Do we want a full inheritance system for default implementations between impls?
    - We do want defaults in interfaces, but no defaulting to implementations in other impls for now; will continue to look for a useful pattern using names somehow
- Also talked about `as` conversions, mostly going to stay with current proposal
    - More than one way to convert float -> int, need a name to distinguish the semantics
- `as` to construct a class with private members -> use case for private impl
    - Concern: behavior in file with access to private impl is different than in other files for the same lookups
    - Other use cases for private impl can be addressed by using a private adapter:
        - private assignment, private destructors for a reference-counted class
    - Maybe `Self` is an adapter with the same fields but public and supporting `as` for construction
    - Example:

```
class X {
  private var a: i32;
  private var b: i32;
```

```
  fn Make() -> X {
    returned var me: X = {.a = 1, .b = 2};
    me = {.a = 3, .b = 4};
    me.a = 5;
    return var;
  }
  fn Get() -> X {
    // 5 options:
    return {.a = 1, .b = 2};
    return {.a = 1, .b = 2} as X;
    return {.a = 1, .b = 2} as private X;
    return {.a = 1, .b = 2} as Self;
    return {.a = 1, .b = 2} as private Self;
  }
}
```

    - Want all of these to work, and none should work outside the class as written (since `a` and `b` are private).
    - Maybe will change impl lookup in a class member to also find impls on the `private self` type.
    - Could impl `private X` as assignable, and then assignment would be allowed in a class member
    - Conjecture: within a class member, `Self` has an implicit `where` clause `Self == private Self`. The body has the `where` clause, not the signature. Doesn't quite work though: `private Self` is an adapter not a facet, but like the basic idea of contextually allowing this lookup
    - Would like to be able to write a constraint that captures whether you have access to `private Self`. Could just make `private` a type qualifier.
    - Maybe: `where` clauses make more impls available; private impls are examples of impls that can be made available by `where` somehow.
    - Motivating use case is calling a generic with impls not available in public API -- but this can be solved using private adapter types.
    - Perhaps we should just abandon trying to support delegation of private access.
    - Instead, give up on the customizability of `as` when converting from a struct type to a class type. Possibly this could be restricted to just when the struct has the same field names as the class.
    - Conclusion: users are forbidden from defining an impl for `As` struct -> class type. Compiler automatically defines struct -> class implicit conversion as part of class definition, and only makes it available in class methods and friends if any data member is private.

## 2021-10-18

- Attendees: chandlerc, josh11b, jonmeow, zygoloid
- [josh11b] New generics constraints proposal, still has some open questions on syntax and naming

- [josh11b] Do we have everything we need for operator interfaces?
    - Have: associated types, interface parameters
    - Still to come: parameterized impls, not sure if they are blocking
        - Find requirements on parameterized impls by seeing what we need for operator interfaces
    - To decide: Should the library "add with Carbon's built in integer types" impl be parameterized to add with anything that can implicitly convert to an int?
        - Probably better to just pick "yes" for now since that is one of the anticipated use cases for supporting implicit conversions, and we need experience to determine how that goes wrong
    - Rust uses a default of =Self for the RHS and return type. Concern that is weird for Carbon.
        - Concern is what Addable() means outside of a context where `Self` is known
        - Could make a named constraint that extends Addable, setting RHS to `Self` if we don't want to allow that
    - Question: If an interface only has parameters with defaults, do you have to say `()` after it?
        - Provisionally yes, to be consistent with other function calling; contrary to Rust which uses `<...>` to mean something a little different than a function call.
        - If you want to add parameters to an interface that didn't start with them, can make the parameterized thing have another name, and then use a named constraint to make an alias for the thing with no parameters
        - Need type aliases to do the same thing for parameterized types
- [zygoloid] Would like a better story for arithmetic overflow
    - [chandlerc] Can solve this later
    - [zygoloid] From last week: integer types overflow behavior
    - claim is that most uses of unsigned integers want overflow to be an error
    - [zygoloid] Perhaps we have a modular type
        - [josh11b] Concern is that hashes still may want to use division
    - [chandlerc] Concern about the data from doc: question is about unsigned types, but doc mostly covers signed types
- [zygoloid] Discussion about top-down vs. not top-down name lookup and info propagation
    - [josh11b] Both small poll and recent language designs agree on not top-down
    - [chandlerc] Want to keep a lot of consistency with C++
    - [josh11b] Thinks more important to do what the majority of C++ programmers want
    - [chandlerc] Not going to make-or-break people's opinion of Carbon
    - [chandlerc] People go to the source code in practice instead of generated documentation, even in languages like Java with good generated documentation tooling
        - Definitely cases where people do use generated doc: Python, TensorFlow; so we are going to need both
    - [zygoloid] Forward declarations are complicated, but worth it

    - [zygoloid] Concern is corner cases
    - [chandlerc] Expect to solve corner cases using local fixes
    - [chandlerc] Thinks local fixes will generally be improvements overall
        - Example: overloaded functions need to be forward declared
        - An overload set is like a function, can only be called after it is declared
        - [chandlerc] lexically together?
            - [zygoloid] would be awkward for some existing C++ cases, but maybe those would be replaced by interfaces in Carbon
- [chandlerc] Destructors?
    - Would be fine with (a) C++ model or (b) using partial types, so you can only call methods that are declared as compatible with destruction, no virtual calls
    - Need something, could change it later
    - Generic functions should be able to say whether they only take Type or can possibly call the destructors
    - Leaning toward destructors are in an interface
    - Means we want to support `external` impls defined inside the class scope
    - Interfaces that correspond to syntax / operators should generally be external
        - Still want a nice name for the associated type return type
    - Don't want to support overloading via implementing the same parameterized interface internally twice with different parameters
    - Parameterized interfaces default to external?
    - Perhaps in the future: Parameterized impl -> can inject the name of the interface into the type with additional parameterization matching the parameterization of the impl
        - Only for functions where the parameters of the interface can be deduced from the parameters of the function
        - Basically means the function is overloaded, but only for a subset of the functions
        - Non-function entities would never be visible, since no parameters to deduce from
    - Parameterized impls are always external
    - Would be fine saying only language-defined interfaces for operators, such as assignment, implicit conversions, and destructors, are external even when declared internally
    - Maybe in-scope impls have privileged access, whether they are internal or external, and out-of-scope impls only have public access
        - Chandler thinks this may be an orthogonal access control mechanism
    - Do we want more access control boundaries within a class?
        - Maybe can be supported via inheritance? Right now C++ users don't in practice
        - Actually adapters support this better?
- [zygoloid] Concerned about Self type changing between methods for access control
    - Maybe have private facet only for delegation, otherwise use a rule like C++ where some members can only be used from some scopes

## 2021-10-14

- Attendees: jonmeow, josh11b, zygoloid, jorgbrown
- [josh11b] F# has a mechanism for customizing what happens in a named blocks, used for async (and maybe queries?), a bit like Haskell's `do`, see: Computation Expressions - F#
- [zygoloid] Which combinations of signed/unsigned and overflow-errors/overflow-wraps should we have?
    - C/C++ has: int signed&overflow-errors, unsigned unsigned&overflow-wraps, plus some types that mix overflow-errors and overflow-wraps like signed/unsigned char/short
    - Rust has: iN signed&overflow-errors, uN unsigned&overflow-errors, Wrapping<iN>/Wrapping<uN> for signed/unsigned&overflow-wraps
    - C# has signed types, but overflow behavior depends on whether in a checked or unchecked scope, not the type
    - Java has: int signed&overflow-wraps
    - Idea/proposal: iN signed&overflow-errors, uN unsigned&overflow-errors, and mN "modular" that wraps but is neither signed or unsigned, so doesn't support `<`, `/`, or `>>` operations
    - Would support the following conversions and casts:
        - uU -implicit> iI if U < I
        - iI -implicit> mM if I <= M
        - uU -implicit> mM if U <= M
        - iI -as> mM for all I, M
        - uU -as> mM for all U, M
        - mM -as> iI if I <= M
        - mM -as> uU if U <= M
    - iI models $\mathbf{Z} \cap [-2^{I-1}, 2^{I-1})$, uU models $\mathbf{Z} \cap [0, 2^{U})$, and mM models $\mathbf{Z} / 2^{M}\mathbf{Z}$
    - Can round-trip from iI or uU to mM if the modular type has at least as many bits
    - Swift and Zig model the difference between overflow-errors and overflow-wraps not in the type but in the operation (e.g. + vs. +%)
    - [josh11b] Two candidates seem the most appealing:
        - Like C/C++ with iN signed&overflow-errors, uN unsigned&overflow-wraps
        - As proposed with iN/uN overflow-errors, and Modulo(N) with modular/wrapping but no signedness
            - Modulo(N) for hash & crypto use cases, not common enough for mN shortcut
            - uN for bit packing in cases where there is no room for a sign bit
- [josh11b] What about u31 being used for the type of an array index or array size?
    - [josh11b] Goal is to reflect constraints in the type system so generics can't fail
    - [josh11b] Recall earlier situation where objections to switching on type
    - we agree: overflow when adding generic integers => a monomorphization error
        - More generally: constant evaluation failure in any generic argument
    - [zygoloid] Didn't we say overlap due to specialization could also lead to a monomorphization error?

    - [josh11b] Specialization plan is not resolved. Fundamental issue is deciding what to do when multiple blanket impls apply, as in:
        - BigInt + castable to CommonInt
        - Castable to CommonInt + LargeInt
    - [zygoloid] Any array taking >= 2^55 bytes leads to a monomorphization error, not just negative sizes
    - [josh11b] Perhaps we should only allow generic + on BigInts, not iN?
    - Should array size be a template parameter? That way it would be legal to do a range check and fail instantiation
    - Aren't array sizes the main use for non-const generics? If those won't be generic, what else are non-type generics used for?
        - integer bit size
    - Specialization should require TypeId capability
        - if Type, only need a single instantiation
        - if Sized, only need an instantiation per size
    - Other use for generics is supporting dynamic dispatch, which is awkward with specialization
    - Goal:
        - templates => instantiation can fail
        - generics => monomorphization succeeds. Constraints that could cause failure are pushed to the consumer. True for types *and* non-types.
    - Conclusion: We only allow operations that are total to produce a generic value
        - so most operations result in runtime/dynamic values, even if the arguments are generics. Example: + on two generic i32s produces a runtime/dynamic value.
        - the things that can produce a generic value are:
            - non-user-defined type constructors, like postfix-* and parameterized types
            - member access
            - as expressions converting to facet types
            - all things built into the compiler, no user-defined operations, so compiler can validate be assured they are total
            - overlap with things the compiler can invert so we can support deduction with, but not the same
    - Conclusion: we support a way for templated types to implement a generic interface, so they may be used by generics. Generics type check against the interface, but the template instantiation is allowed to fail.
        - Model is that there is an implicit passing of the implementation of any template or specialization used transitively for a type along with the witness table of that type to the outermost generic call
        - May be tricky to support that directly with separate compilation, so instead errors here may be generated late
- [zygoloid] What about the Modulo(N) vs. C/C++ question?
    - [josh11b] Would like to see how common the use cases are. If most of the unsigned use cases are cases with wrapping, would prefer the C/C++ approach. Concern is proliferation of types when wrapping is either correct or irrelevant for

most unsigned use cases. If most unsigned use cases don't want wrapping behavior, a separate Modulo(N) type makes more sense.

## 2021-10-12

- Attendees: josh11b, geoffromer, wolff, gribozavr, zygoloid
- Trying to make programmers guide following by Rust-by-Example
    - [wolff] Assembling a spreadsheet with language questions to answer to fill this out
    - [wolff] How much can we get by with just interop? E.g. maybe file I/O could be done through C
- [gromer] Executable semantics: How can we structure the implementation of name resolution and type checking to be resilient to changes in Carbon's design?
    - Originally was going to do name lookup during parsing
    - [josh11b] Separate passes? Or are the passes separable?
    - [geoffromer] Type resolution likely depends on name resolution, but name resolution might also have to depend on type resolution for e.g. out-of-line method definitions unless everything is qualified using `Self`
    - [zygoloid] Can push the language design to make this easier if we want to. Complex trade-off space, not going to be an easy decision. Seems like we will end up with something like C++ where you can re-enter a class' scope.
    - [josh11b] Flexible design for executable semantics so it doesn't depend on this decision?
        - [zygoloid] Have unresolved identifier nodes get converted to resolved at some unspecified point
        - [geoffromer] Only additive mutations unless you are making a deep copy of the whole AST
        - Do AST nodes have parent pointers? Not at the moment, but some kinds of up edges, e.g. break, continue, and return nodes
        - [josh11b] Could name resolution be an optional part of an identifier node so it could be resolved additively?
            - [geoffromer] Concern is once name resolution is done, envisioning having a separate table with objects that represent each symbol, so AST nodes that have names are changed to point to the appropriate symbol. It is additive, but does introduce new nodes which is complexity we may not want. New nodes would likely not be AST nodes per se, but have considered both ways. Problem with names pointing to AST nodes is things like overload sets that don't have a single unique AST node representation.
    - [zygoloid] How do we want unqualified name lookup? Spectrum from top-down approach to global availability of symbols.

- [geoffromer] Doesn't like either extreme. Would find it confusing if a later local variable declaration can affect prior code in the function body like in C#. A purely lexical name lookup rule seems like it would impose a lot of awkward constraints without a lot of user benefit. Hunch is close to C++ but hopefully more coherent and principled
- [josh11b] What sort of data would help make this decision? Seems like this decision will be baked pretty deeply into the compiler
- [gribozavr] Going to have way more users than compiler implementers
    - [geoffromer] There will also be tooling authors
- [geoffromer] If we can keep name lookup a separate phase prior to type checking, could be a big enough conceptual simplification to be worth entertaining. Re: data, we might just need to try it and see how onerous is it to qualify everything in method definitions
- [josh11b] Not just a compiler/tools issue, but a code reader vs. code writer issue
- [zygoloid] Any recent languages that use a top-down single-pass approach? Could leverage their existing research?
    - [geoffromer] Maybe Go? https://golang.org/doc/faq#principles "There are no forward declarations and no header files; everything is declared exactly once." A guiding principle of the design of the language to avoid that bookkeeping and repetition.
    - Python is weird; names in a function have special treatment if they are assigned later in the function. Python mostly behaves as if everything is visible everywhere
- [geoffromer] Re: pain points, in personal experience as a writer it is an occasional annoyance to have to shuffle things around. As a reader, doesn't get much value from the upward looking rule; in class scope there generally isn't an order and haven't had problems
- [zygoloid] There is a fundamental difference between different scopes, some scopes execute code sequentially have an inherent order, others the ordering seems arbitrary.
- [geoffromer] The order of data member declarations is significant, but it doesn't seem very related to name lookup. Do agree with the basic point that it makes a lot of sense to treat function bodies differently.
- Are the cases: functions, classes, and namespaces? Interfaces and choice types the same as classes; package scope like a namespace.
- Seems like no recent languages use top-down single-pass.
- [wolff] What about circular references?
    - They happen and need to be supported
    - [zygoloid] Circular references can arise from templates no matter what, so will need to be diagnosed regardless of what name lookup rule we use

        - Problems result from trying to instantiate a type when it is currently being instantiated
    - [geoffromer] Verbiage in C++ standard about when things are instantiated is confusing today. But doesn't affect language users as much. Would be nice if Carbon didn't have that complexity.
    - [zygoloid] Comes up when implementing Variant, Any, or something. It does make sense to say if you as a question of a type and it can't answer, it should be a hard error
        - [geoffromer] a relatively recent adopter of IDEs; the important question is what the IDE user's experience is going to be like not how hard it is for the IDE implementer
        - [geoffromer] Easier to implement an IDE => good for adoption
- Autocomplete while code is being written?
    - [zygoloid] Not innovative, lots of languages have already solved this problem
    - [gribozavr] Solved this for Swift in 3 months as an intern at Apple. Type checking was trickier, such as dealing with generics, because more reliant on the code being correct. In a C++ IDE, seems better if you are at the top of the file to suggest things later in the file and add the forward declaration if it is needed.
- Have consensus among the people here, wait for Chandler to resolve

## 2021-10-05 part 2

- Attendees: geoffromer, dhollman, zygoloid, gribozavr, chandlerc, josh11b
- [josh11b]: Rewrite approach to executable semantics?
    - [geoffromer] somewhat in conflict with mutate-in-place approach without big structural changes
    - [chandlerc] additive good, otherwise immutability of AST is valuable
    - [chandlerc] C++ standardization folks really intensely dislike rewrite rules, but that is specific to C++; all proposed rewrite rules have turned out to be wrong in some way
    - [chandlerc] People who teach C++ do explain it using rewrite rules, "the way this works under the hood is as if <rewrite rule description>", even though they are not precisely correct
    - [zygoloid] Came up recently with range-based for loop, variable is not in scope despite where it would end up in a rewrite
    - [chandlerc] Carbon team has been describing things in terms of rewrites. Controversial idea is to embrace this.

- Model is to embrace rewrite rules in a specific way, where we actually insist that the rewrite we describe is 100% accurate. If it isn't we don't describe it with a rewrite rule. We would have a fairly rigid understanding of the specific before and after language dialects. Two additional dialects so far
- One is a syntactically lowered, where lowering does not use type information, "desugared representation". Operators and other part of the expression grammar would be rewritten at this stage. Fewer and more orthogonal primitives
- Resolved representations after all type system resolution. Resolved: overloads, implicit conversions, etc. Type checker of this representation only needs to check type equality.
- We don't want the rewriting-related concerns to distort the design of language that users use.
- Would like the clarity of understanding whether a new construct is a new primitive or gets rewritten out
- Would like the the result of the rewrite to be legal Carbon, could be pasted into the source code input and get exactly the same semantics
    - Would need some additional primitives that wouldn't be expected for users to use
    - Example: `*p` desugared to `p.(Ptr.Deref)().__DEREF__` where `__DEREF__` is something that can turn raw pointers into l-values; no way to keep people from using it
- [geoffromer] Cautionary tale: range-based for loops, the basic lifetime issue of temporaries to the right of the colon; arguments on both sides about whether that's a bug that results from the specification as a rewrite. This issue would have been noticed earlier by more people if it were not effectively hidden in the rewrite. It seems like this approach would be prone to those hazards.
- [zygoloid] Do really like modelling semantic steps as a sequence of incremental rewrites. That makes good sense, it is a well studied approach. Another concern is we very likely want different semantics for the lowered language. For example, no implicit conversions.
- [gribozavr] The l-value issue is a good example. Swift has had problems trying to expose this, because it isn't safe without a Rust borrow checker.
    - or destructor calls
- [chandlerc] Would absolutely want to have disjoint syntax whenever there is disjoint semantics, just for clarity. So the l-value forming thing should absolutely not use `*`.
    - Would like a Carbon syntax for avoiding implicit conversions anyway.

        - Maybe accepting the rewritten language would require a debug flag? Maybe okay unless we can't support mixing the two languages.
    - [josh11b] I'd be more supportive if this was somehow not documented. some escape hatch in the compiler or something
        - Useful for people trying to propose features
        - Would not be legal to check in, would be rejected unless opted in
        - Otherwise this creates a lot of language design constraints that don't seem beneficial and could have a lot of harm
        - [dhollman] But....Hyrum's law?
        - [gribozavr] like perma-unstable rust features
    - [geoffromer] if they're not part of the language as specified, they aren't really useful for specifying carbon? so might defeat the purpose..
    - [geoffromer] creating a syntactic space where implicit conversions can't happen seems like that might be a separate language that would be union-able. implicit conversions happen in so many places. function call syntax, initialization, etc. at that point the value proposition seems unclear
    - [chandlerc] There should be a syntax that is guaranteed to have no implicit conversions, this should at least be doable.
    - [chandlerc] Agree with the language design constraint, if we have this it will get used, has to be restricted. Should not spend time supporting language evolution at this level. No support or stability guarantees for anything with some underbar syntax. Requires a flag. Strongest barrier would be to refuse to union the languages. Would be fine to document it and include it in the specification.
    - [chandlerc] Personal preference: some kind of delimiting syntax to designate regions with the lowered languages; could limit switching to this, e.g. not in the expression grammar, only at a block or statement level; plus maybe a flag
        - [josh11b] my preference would be the flag to default to disallowing
        - [chandlerc] flag would default to allowing
        - [chandlerc] libraries might use this to workaround a bug until a new compiler release is available; sympathetic to wanting more tools for libraries to deal with problems
        - [chandlerc] Example: we currently check in inline assembly at Google to work around bugs; also use some extremely sketch C++ constructs to turn off parts of C++ to work around bugs
        - [chandlerc] More of an engineering-trade-off than a big deal.
        - [josh11b] Maybe a flag that defaults off but a library can turn on for itself?
    - [zygoloid] Any languages that expose a lower-level language?
        - [josh11b] maybe example: MLIR

        - [chandlerc] maybe: Lisps, most embedded DSLs rewrite
        - [josh11b] Scheme?
        - [chandlerc] Originally CFront was defined as a rewrite to C
        - [dhollman] LaTeX lowering to TeX
        - [chandlerc] Scala semantics are defined by a primitive subset of Java which is equivalent to JVM bytecode
        - [chandlerc] Modern Java rewrites to a proper subset of Java
        - [chandlerc] Kotlin includes ability to have Java code
        - [dhollman] CUDA
    - [zygoloid] None of the above are rewrites for the reasons we are considering
        - CUDA rewrite process can go wrong, necessitating bypassing
        - CUDA could only handle what could be rewritten to a host compiler, but not trying to limit to natural translation
    - [dhollman] What is the connection between a rewrite and generative metaprogramming?
        - [chandlerc] Think we will discover when we try to define these lowered languages, I think we will discover useful facilities to expose in Carbon for the purpose of metaprogramming.
        - Might want to have a metaprogram that writes in the language of users, other metaprograms that produce code in a lowered language without implicit conversions and so on
    - [dhollman] Want the rewrite to produce something that is clearly defined but not necessarily readable, similar to a metaprogram or the output of the compiler. Focus is how easy it is to reason about. Could be disjoint or a pure subset. Would like this to be the same as what metaprograms produce.
    - [josh11b] to be clear, we both *speculated* about both use cases, we didn't discuss concrete motivating examples
    - [dhollman] readable rewritten code is a pipedream. code generation to something that is readable is not typically worth the effort. certainly not at the expense of not being able to reason about the code or meta code
    - [gribozvar] reason we couldn't expose L-values as a first class construct is that they aren't safe and the don't fit in the language. but they do fit within the rewrite model. the issue is we don't provide facilities in the language to do their own rewrites. <something I missed>. does seem to be a good connection between metaprogramming and this. these kinds of low-level features don't make sense w/o some user controlled rewriting to make use of them
    - [zygoloid] Primary concern in this space, whatever generative meta programming facilities should produce a semantic representation of whatever its using and not tokens and not whatever the equivalent of the

source form. Then it matters less whether it is a higher or lower level language.
    - [dhollman] What examples do we have?
    - [zygoloid] Show me how you interpreted this code? Many compilers can do this: Swift, Rust, GHC, etc. Useful for user's understanding and particularly for debugging metaprograms. Gives most of the value, maybe not all of it.
    - [chandlerc] So maybe hypothesize what the right trade-off it would be. Utility in at least considering adding to the language proper. Features that would come trying to represent this lowered form. Something which is guaranteed to not trigger implicit conversions.
    - [zygoloid] A problem for library design, relying on implicit conversion so that Foo arguments get converted to Bar invisibly
    - [chandlerc] Can already do this using templates. What would we need to produce a fully resolved expression? We don't need to express that over time, just guarantee that there are not additional conversions when there is already an exact match. Want the ability to select a specific overload and call it, have it in C++ and probably want it in Carbon. Also features that we don't want to expose like l-value formation.
    - [geoffromer] Mentally equating "code with no implicit conversions" and "code that asserts it has no implicit conversions"; first is easy and straightforward the second is very difficult.
    - [dhollman] There is an assumption in this discussion that implicit conversions are a useful tool for refactoring. Maybe not getting the same overload is maybe a feature, maybe we need better refactoring tools and metaprogramming.
    - [zygoloid] Picking a particular overload out of an overload set to mee feels very much like punching through an abstraction boundary; probably reflection could do this exactly, like reflection shouldn't use it most of the time, it is brittle; want an obvious source-level marker for this just like reflection
    - [josh11b] Don't think this provides motivation for including something in Carbon, would only want to see things available in Carbon if they were motivated by a use case of actual users using the language

## 2021-10-05

- Attendees: chandlerc, josh11b
- Discussion about defining Carbon semantics in terms of rewrites
    - Two phases:

        - one that doesn't use types but can produce unqualified method calls and does not resolve overloads, and
        - one that uses types and produces only resolved calls
    - Desire: there exists some Carbon syntax for specifying the specific overload and implicit parameters so that the result of the rewrite is still legal Carbon
        - Might use this syntax in meta-programming where you sometimes want to specify exact types instead of relying on overload resolution
    - Possibly useful for explaining what the compiler is doing
- What are the dependencies on Generics, in particular constraints?
    - Operator interfaces probably don't require constraints
    - Standard library probably will use constraints for e.g. container interfaces
    - Will want an iteration interface for range-based `for`, probably won't use constraints
    - [josh11b@] we will support both `a[3..10]` and `a[5]` by selecting between different interface parameters in the `Index` interface used by the `[]` operator.
        - `3..10` would make an integer range that could be passed into a range-based for as well as `[]`

## 2021-10-04

- Attendees: josh11b, zygoloid, chandlerc
- Generic constraint options:

| | |
|---|---|
| Swift / Knuth-Bendix | Same drawbacks as before |
| Rewrite to arg passing | Broken |
| Arg passing | Broken |
| Emoji algorithm | Broken |
| Regular equivalence classes | Unknown if we can make terminate |
| Restrictions | No candidates yet |
| Manual, only one automatic `where` clause conversion | Verbosity and ergonomic concerns |

- Considering "restrictions" approach
    - Goal is to discover what restrictions are viable and which use cases are needed
    - Could pair with "Swift / Knuth-Bendix", or maybe restrictions themselves support an algorithm
    - Goal would be to have a set of restrictions that:
        - Allows desired use cases
        - Can be evaluated
    - Candidate initial desired use case is this example from Swift (look for "Considering the following definitions"):

```
protocol IteratorProtocol {
  associatedtype Element
}

protocol Sequence {
  associatedtype Iterator : IteratorProtocol
    where Iterator.Element == Element
}

protocol Collection {
  associatedtype Element

  associatedtype SubSequence
    where SubSequence : Sequence,
          SubSequence : Collection,
          SubSequence.Element == Element,
          SubSequence.SubSequence == SubSequence

  associatedtype Index

  associatedtype Indices
    where Indices : Sequence,
          Indices : Collection,
          Indices.Element == Index,
          Indices.Index == Index,
          SubSequence.Index == Index,
          Indices.SubSequence == Indices
}
```

        - Note that this example could be represented by just `.A == B` restrictions with no forward references, but that turns out to not be very restrictive.
        - One idea is to say it has to reach a "steady state" at some defined point:
            - If you are recursive then either equal (`==`) or independent
            - The N-th is like the 2nd
        - Hope is this admits an algorithm that does a finite amount of search since it only has to consider through the point the steady state is established
        - Still hard to make this rigorous when have many associated types that implement the current interface recursively -- `Subsequence` is treated differently than `Indices`
- What about the manual approach?
    - Is this something that would combine / layer on top of another approach to handle cases where it could not tell two expressions name the same type?
        - Not really -- with the Swift / Knuth-Bendix approach we want to reject interfaces that the algorithm doesn't succeed for, so we can have definitive answers for some type questions. This is because if we have

```
X:! A;
```

```
Y:! B where Y == X;
```

we want `Y` to have type `A & B`.

- We want constraints to allow us to give more specific types to the members of interfaces we use. An example:

```
interface Graph {
  let E:! Edge;
  let V:! Vert where .E == E and .Self == E.V;
}
```

- Sad not being able to write this more symmetrically, like `where V.E == E and V == E.V`.
  - We need some approach that works for types that don't have names, can add more later.
- If we allowed forward reference, could write a different symmetric approach:

```
interface Graph {
  let E:! Edge where .V == V;
  let V:! Vert where .E == E;
}
```

- How would that example work with the manual approach to generic constraints?

```
interface Vert {
  let E:! Type;
  fn EdgesFrom[me: Self]() -> Vector(E);
}
interface Edge {
  let V:! Type;
  fn Head... -> V;
  fn Tail... -> V;
}
interface Graph {
  let E:! Edge;
  let V:! Vert where .E == E and .Self == E.V;
}

// Return the number of vertices reachable
// by following edges from `start`.
fn Reachable[G:! Graph where G.V is Hashable](g: G, start: G.V) -> i64 {
  var h: HashSet(G.V);
  var q: Queue(G.V);
  q.Insert(start);
  while (not q.Empty()) {
    var v: G.V = q.Dequeue();
    if (h.Has(v)) { continue; }
    h.Insert(v);
    // auto is deduced to be `G.V.E` which is `Type`, does not have a `Tail`
```

```
method.
    // Could follow one `where` and discover `G.V.E == G.E` which is `Edge` and
    // has a `Tail` method, but that would make the code brittle to also finding
    // another type that also implements `Tail` (like `Cat`).
    // Code works fine if `auto` is replaced by `G.E`.
    for (var e: auto in v.EdgesFrom()) {
      // If we allow you to call `Tail` on a value of type `G.V.E`, then the result
would
      // have type `G.V.E.V` which can't be converted to `G.V`, the type accepted
by
      // `q.Insert(...)`, with a single `where`. If `e` is given type `G.E` instead
of auto
      // then e.Tail() is legal, returns a value of type G.E.V which can be
converted
      // to G.V for q.Insert(...) using a single where.
      q.Insert(e.Tail());
    }
  }
  return h.Size();
}
```

- `v.EdgesFrom()` returns a value of type `Vector(G.V.E)`. The range-based-for will cast that to an iteration interface, as in `Vector(G.V.E) as Iterable`, and then `e` will be initialized from a value of type `(Vector(G.V.E) as Iterable).Element`. `Vector`, to be useful from generic code, needs to be able to promise that `(Vector(T) as Iterable).Element == T` for all `T`, even in the presence of specialization of `Vector`. This equality is a fact about the type of `Vector(T)`, *not* the result of a `where` clause and so does not count toward the "one where clause" searching limit.
- `e: auto` gives `G.V.E`, which has type `Type` and does not have a `Tail` method, requires a cast to `G.E`. But `e.Tail()` returns a `G.E.V` which can be implicitly converted to `G.V`, the parameter type of `q.insert`.
- Could use all `where ... is Interface` clauses, but only one hop of `where ... == ...` clauses
  - Would avoid the need to have additional `observe` clauses for interfaces, as in `observe G.E as Cat == G.V.E as Cat;`
  - If we did require those `observe` clauses, it would be enough to allow `G.E as (Cat & Edge)` without further observations.
- This would result in more explicit types more than it would result in more casts
- Does it matter whether `HashSet` is declared `class HashSet(T:! Type where .Self is Hashable)` or `class HashSet(T:! Hashable)`?
  - Biggest concern is what `h.Front()` returns, want the answer to be `G.V` in both cases, not `G.V as Hashable`
  - Just like `G.V` and `G.V as Hashable` are facets of each other, `HashSet(G.V)` and `HashSet(G.V as Hashable)` are facets of each other. The difference between them is whether they return `G.V` or `G.V as Hashable` in the methods.

        - The distinction would only affect the definition of the body of the methods of the class, not users of the class
        - Same for functions as classes
        - Simpler to explain if this behavior affects the parameter types in addition to return types, even though this may not be observable
        - `-> T` gives you the caller's `T`, `[T:! Container](...) -> T.Element`, get caller's `T.(Container.Element)`
    - Property we would like for `observe`: Strong version: it can only make an invlaid program valid, it cannot change the meaning of a previously valid program. Essential version: ... that is still valid.
        - Imagine we have two possible types transitively reachable from `where` clauses: `Cat` and `Edge` that both have a `Tail` method. Concern is what happens if an `observe` can make both visible when only one was before at some site calling `Tail`.
        - Essential version still holds regardless as long as the ambiguity results in error rather than a change in which `Tail` method is called.
        - If `observe` only added interface implementations externally, then would get the stronger property.
- Standard library
    - Print functions
        - `Log` would interact with the libc file buffering, used for both "Hello World" or printf-debugging, maybe via a separate `Debug` that uses `Log`, but includes time, file, line, etc.
        - Separate StdIn/StdOut for dealing with files piped at the command line
    - Formatted strings can produce streams
        - can be converted to a string easily
        - can be consumed by Log and file IO to use the existing buffer
    - Put enough (`Log`) in the prelude for "Hello World"
        - Maybe: Log(Print, "Hello World")
    - Maybe have hooks in logging to handle Google-scale logging facilities
    - Where does --help output go? Print
    - Will design translatable strings separately from other string formatting
        - Experts will have specific ideas about what to do with translatable strings
- Want `T:! Type` to give no capabilities so that we can always have a single instantiation. Means not just unsized, but not destructible!
    - Maybe have some applications for types that are not destructible. Perhaps we want to support linear types that can only be destroyed by becoming unformed as a result of being moved?
- What are our basic interfaces? Here `->` means "implies"
    - Copyable -> Sized -> Type
    - Movable requires HasUnformed, but Copyable doesn't
    - Movable -> HasUnformed -> Sized
    - Copyable & HasUnformed -> Movable
    - TriviallyCopyable & TriviallyDestructible & HasUnformed -> EfficientlyMovable
    - Maybe:
        - Copyable & TriviallyDestructible -> Relocatable?

        - Movable -> Relocatable -> Sized
    - No, instead:
        - Copyable -> Relocatable -> Type
        - Relocatable does not imply Sized
    - TriviallyCopyable -> TriviallyRelocatable, meaning "can be memcpy-ed"
    - Contrast with EasilyRelocatable, which means it has a post-memcpy fixup. To be useful for `realloc` use cases, may be given old pointer or pointer offset, but old pointer will be invalid to access
    - Types, Type of Types, function values are *not* sized and so can't be used with `var`. May also not be destructible.
    - ~~literals are not storable, so irrelevant if they are sized (possibly they are 0-sized), they can't be used with `var` or be the pointee of a pointer type~~

## 2021-09-28

- Attendees: josh11b, zygoloid, chandlerc, jonmeow, geoffromer
- Review of yesterday's meeting
- The intermediate vertices in a DFA correspond to types that we know something about, but that aren't the same as the equivalence class that we are defining.
    - The accept vertex will be the equivalence class we are defining
    - For `(A|B)C`, `A|B` may not be an equivalence class, but it is an expression that we know more about
- Applying the completeness rule to itself: if it repeats, have the commutator case that is problematic
    - don't know if can also fail to terminate via mutual recursion
- Doesn't rule out `X = YXY` case that isn't regular
    - would match `YXYXY` in two overlapping ways: `XXY` and `YXX`
    - More generally there is a concern with `Z=YXY`
    - What happens when you complete it with itself?
        - YXYXY|YXZ|ZXY
        - YXYXYXY|YXYXZ|YXZXY|ZXYXY|ZXZ
        - infinite family of equivalence classes, not regular
        - so forbid it
- Bool -> int conversion
    - true -> 1, false -> 0 is a bit arbitrary, e.g. true <-> -1 is also plausible
    - do we want to allow ints as the arguments to `if`, `and`, `or`, and so on?
        - No, benefit from being explicit boolean, experience is that is error prone
        - People use that conversion more commonly than wanting to convert a bool -> int
    - Something about nullable pointers and optional values
    - Implicit conversion bool -> int problematic, but no history of bool -> int explicit conversion bugs
    - [geoffromer] Works farther from the hardware, experience is treating it as a choice type, where the specific value is not important
        - minor readability stumbling block in code that converts bools to ints

        - doesn't have a better spelling that doesn't involve control flow, concedes that this operation is still desired/useful
        - if there was a good name for a function, would be better, but best is "ToInt" which is basically the same as "as Int"
    - Use cases:
        - [zygoloid] count number of trues in a set of bools
        - [chandlerc] turn a bool into a mask
    - Unsurprising, conventional to use 0, 1, even if it isn't obvious
    - `AsBit`
    - [chandlerc] For hardening, want zeroed memory to be treating bools as false; similarly zeroing will clear flags
    - [zygoloid] Analogy with zeroing pointers is that we don't necessarily want a correspondence between pointers and integers
    - [geoffromer] Uncomfortable with converting a bool to an int roughly to the same extent and for roughly the same reasons uncomfortable with using integer types for bit smashing
        - [geoffromer] Withdraw objections to bool -> int conversion, unless and until we get a dedicated bit smashing type
    - [chandlerc] Stepanov's description of what makes C++ special, would like to keep in Carbon:
        - exposing the bits of the hardware to the user
        - the addresses of objects is exposed
        - building abstractions on those two
    - Maybe can make the bit operations a bit more user friendly, but definitely like exposing that integers are represented as bits in 2s-complement, etc.
    - Exception: special x86 instruction MOVMSK that takes the *high* bit of a a simd vector and compresses them into the bits of an integer
    - [geoffromer] Conversion from bool -> int skips a step, should go through a bit first
        - [chandlerc] `u1`, the canonical type for a bit, does implicitly convert to any integer type
    - [zygoloid] what is the intent `uN` type?
        - [chandlerc] a sequence of bits modelling an integer
        - Is the byte type the same thing as `u8`? there are other types that represent the aliasing or poisoning properties when reading memory
    - [geoffromer] Maybe we should restrict conversion from `uN` to signed integer types
        - [chandlerc] no principled reason, just utilitarian and following the history of C & C++
        - [chandlerc] if we allow shifting, bit-oring, masking, etc. on `uN` types, then they are really collections of bits
    - [zygoloid] started with allowing explicit conversions in both directions
        - but now not convinced by the utility of the bool -> int conversion
        - [chandlerc] would be horrible to rewrite code that turns conditions into bitmasks
        - [zygoloid] actually uses (condition) ? NOT_JUST_ONE_MASK : 0 in practice

        - can't currently justify having just bool -> int without int -> bool
        - revisiting this: rationale is that we have an obvious and unsurprising embedding from bool -> int that preserves bit operations (there are others like false -> 0, true -> -1, but that would be surprising to some); on the other hand it's not obvious in the same way what int -> bool should do: `0 as bool` and `1 as bool` seem fine, but what is `2 as bool`? any answer seems like it would be surprising to some.
- Back to generic type checking
    - Don't know if completion will terminate
        - Could have the restriction that every equivalence class corresponds to a where clause
        - MAYBE: If completion identifies overlap that would create a new rule, error
        - eliminates overlap that isn't containment
    - What about the compatibility rule?
        - about containment
        - Does containment form a partial order?
            - No, no containment in the edge/vertex/graph case
        - edge/vertex/graph case is a graph with a start point and two other vertices, both of which accept different equivalence classes
        - intuitively terminates, but no obvious reason yet
    - Question: can we get infinite completion between two different rules without ever having a single rule be self-incomplete?
        - `A(BC)* (CB)*D`

## 2021-09-27

- Attendees: josh11b, zygoloid, chandlerc, jonmeow
- [josh11b] Comparison of generic constraint type checking algorithms

| | Terminates | Correct | Expressive | Clear what's legal | Easy to say what you want |
|---|---|---|---|---|---|
| Swift/Knuth-Bendix | Undecidable, uses a step limit | Yes | Yes | No | Yes |
| Argument passing | Yes | Probably | Medium | Yes | Low |
| Rewrite to arg passing | Yes | ? | Medium | Low | Yes |
| Emoji algorithm | Yes | ? | Yes | ? | Yes |

| | | | | | |
|---|---|---|---|---|---|
| New: Regular | Still working out details | Yes | Low, but likely enough | Yes | Yes |

- [josh11b] Argument passing approach has a terminating type equality algorithm, but relies on type checking which hasn't yet been fully specified to validate that uses of `Self` and `.Self` are okay
- [josh11b] Argument passing approach relies on user-provided order without forward references, rewrite to arg passing has to invent this order, which hasn't been fully specified yet
- [josh11b] Emoji algorithm approach emerged from trying to use type checking to validate an interface declaration independent of forward references, to remove limitations that are more awkward to understand in the `where` syntax than the argument passing syntax
- [joshl11b] Emoji algorithm was being worked out in this doc, works by putting each interface or function declaration in a normal form. Normal forms may be composed by treating them as directed graphs and overlaying rooted at different points using graph traversals
    - Main idea is that you run the type canonicalization algorithm used at query time "enough" at interface type checking time that any problems would be detected
    - Hope was that we could type check each interface in isolation and the composition would be good by induction
    - Problems with the algorithm are related to the results being sensitive to traversal order so danger of inconsistency between different queries and between queries and type checking
    - Tried to make changes to type checking to detect situations where it was sensitive to order and reject, but tricky to validate that is sufficient and tricky to characterize the situations where it will reject
    - "Commutator" relations (`X.Y == Y.X`) seem particularly problematic
- [josh11b] New algorithm starts with the motivation that we are willing to sacrifice expressivity in favor of making it clear what is legal since there are relatively few patterns we have use cases for
    - Important to support rewrites like `X<->X.Y` and `Y<->X.Y` but not `X.Y<->Y.X` which seems to be a source of trouble
    - Simplifying assumption for now: all associated types have type that is recursively the same interface, and have 1 letter names
    - Simple example: let's say we have a rule like `X<->Y`. Replace every occurrence of `X` or `Y` in queries and rules with a new symbol α.
    - Consider these two rules: `A <-> A.B` and `B <-> B.C`. Can we rewrite `A.C` to `A`? Yes, but only by making the sequence longer before making it shorter: `A.C -> A.B.C -> A.B -> A`. Not clear how to find this rewrite.
    - Solve this problem by considering this harder problem: can we characterize the entire equivalence class of sequences the rewrite rule can rewrite between.
    - In many cases, the equivalence classes are regular languages, and can be represented by regexps.
        - For the `X<->Y` example, the regexp is `(X|Y)`, which we can use in place of the symbol α.
        - `A <-> AB` => `AB*`
        - `A <-> AB` + `B <-> BC` => `A(BC*)*`, `BC*`

    - Problem cases are commutators
        - `XY <-> YX`
        - `C <-> AB` and `C <-> BA`
        - `ABC <-> CBA`
        - These require an infinite family of regular expressions
        - So we'll exclude these
    - Only using *, or, and concat
        - `B <-> ABA` would require back references, out of scope
    - Example with multiple interfaces:

```
interface Graph {
  let E:! Edge;
  let V:! Vertex where .E == E and .Self == E.V;
}
```

Want to combine `E.(Edge.V) | V` and `E | V.(Vertex.E)`. Substituting gives:

```
(E | V.(Vertex.E)).(Edge.V) | V
-> E.(Edge.V) | V.(Vertex.E).(Edge.V) | V
-> E.(Edge.V) | V(.(Vertex.E).(Edge.V))*
-> (E.(Edge.V) | V)(.(Vertex.E).(Edge.V))*
-> (E.((Edge.V).(Vertex.E))*.(Edge.V) | V(.(Vertex.E).(Edge.V))*
```

    - Rules with partial overlap, such as `d <-> ab` and `e <-> bc`
        - Need to create a rule for the union in the case they overlap
        - `(abc|ae|dc)`, `(d|ab)`, `(e|bc)`
        - Could forbid this case, but seems supportable with additional code
    - Goal: accept constraints where there are a finite number of equivalence classes each of which is a regular language
    - Achieve soundness with three things:
        - 1. Compatibility: for `A, B` in `R`, a set of regexps, are compatible if for every string `S` matching `A`, with a substring `SB1` matching `B`, then (`S` with `SB1` replaced by `SB2` also matching `B`) also matches `A`
        - 2. Completeness: for `A, B` in `R`, a set of regexps, `S` a string, where `A`, `B` match overlapping substrings of `S`, *then* there exists `C` in `R` matching the union of the substrings with `C` compatible with `A` and `B`
        - 3. Finiteness
    - Can achieve compatibility by glueing in the DFA for `B` into its match in the DFA for `A`, and then resolving having two edges with the same label exiting a node using the NFA->DFA algorithm: https://photos.app.goo.gl/VcZwJNtCP7BmLRzG6
    - Not all regexps correspond to equivalence classes
        - `(AA* | B)` does not correspond, since `B -> AA -> BA` and `B->AA->AB` so the equivalence class is actually `(A|B)*`
        - In general, all accept states will be a single node
    - Unifying the accept states is a special case of stitching a regexp's DFA into itself to make it compatible with itself, we actually need this more general operation.

## 2021-09-20

- Attendees: josh11b, zygoloid, chandlerc, jonmeow
- [speaker] Comments
    - More comments
- Looked at the archetype algorithm, in particular where clause rewrites
    - Goals are: 1) Correct, 2) Terminates, 3) Expressive, 4) Ergonomic
    - Simple example

```
interface Container {
  let Elt:! Type;
  let Iter:! Iterator where .Elt == Elt;
  let Slice:! Container where .Elt == Elt and .Slice == .Self;
}
```

is mechanically rewritten to:

```
Container
* $1 :! Type
  - Elt as Type
* $2 :! Iterator
  - Iter as Iterator
* $3 :! Container
  - Slice as Container
* $0 :! Container{.Elt = $1, .Iter = $2, .Slice = $3}
  - Self as Container

Iter where .Elt == Elt
Slice where .Elt == Elt and .Slice == .Self;
```

and then `where` clauses are rewritten one at a time in order to get:

```
Container
* $1 :! Type
  - Elt as Type
  - Iter.Elt as Type
  - Slice.Elt as Type
* $2 :! Iterator{.Elt = $1}
  - Iter as Iterator
* $3 :! Container{.Elt = $1, .Slice = $3}
  - Slice as Container
  - Slice.Slice as Container
* $0 :! Container{.Elt = $1, .Iter = $2, .Slice = $3}
  - Self as Container
```

    - Recursing in the `__combine__(A{.X=S}, A{.X=T})` case is desirable for this example:

```
Z:! A where .Y is Comparable;
X:! A where .Y is Printable;  // and we don't want to have to say "and .Y == Z.Y"
W:! B where .V == Z and .V == X;
```

- Everything looks finite, seems like we should be able to do all rewrites, even `__combine__(A{.X=S}, A{.X=T})` in finite number steps
- What goes wrong with `Impossible`? Type checking?

```
interface Impossible {
  let A :! Impossible;
  let B :! Impossible;
  let C :! Impossible where A.C == C.A and B.C == C.B;
  let D :! Impossible where A.D == D.A and B.D == D.B;
  let E :! Impossible where C.E == E.C.A and D.E == E.D.B and C.C.A == C.C.A.E;
}
```

Partially converted to normal form:

```
Impossible
* $1 :! __combine__(__typeof__(A.C), __typeof__(C.A))
  - A.C
  - C.A
* $2 :! __combine__(__typeof__(B.C), __typeof__(C.B))
  - B.C
  - C.B
* $3 :! __combine__(__typeof__(A.D), __typeof__(D.A))
  - A.D
  - D.A
* $4 :! __combine__(__typeof__(B.D), __typeof__(D.B))
  - B.D
  - D.B
* $5 :! __combine__(__typeof__(C.E), __typeof__(E.C.A))
  - C.E
  - E.C.A
* $6 :! __typeof__(E.C){.A = $5}
  - E.C
* $7 :! __combine__(__typeof__(D.E), __typeof__(E.D.B))
  - D.E
  - E.D.B
* $8 :! __typeof__(E.D){.B = $7}
  - E.D
* $9 :! __combine__(__typeof__(C.C.A), __typeof__(C.C.A.E)){.E = $9}
  - C.C.A
  - C.C.A.E
* $10 :! __typeof__(C.C){.A = $9}
  - C.C
```

```
* $11 :! Impossible{.C = $1, .D = $3}
  - A
* $12 :! Impossible{.C = $2, .D = $4}
  - B
* $13 :! Impossible{.A = $1, .B = $2, .C = $10, .E = $5}
  - C
* $14 :! Impossible{.A = $3, .B = $4, .E = $7}
  - D
* $15 :! Impossible{.C = $6, .D = $8}
  - E
* $0 :! Impossible{.A = $11, .B = $12, .C = $13, .D = $14, .E = $15}
  - Self
```

Josh thinks this just becomes:

```
Impossible
* $1 :! Impossible
  - A.C
  - C.A
* $2 :! Impossible
  - B.C
  - C.B
* $3 :! Impossible
  - A.D
  - D.A
* $4 :! Impossible
  - B.D
  - D.B
* $5 :! Impossible
  - C.E
  - E.C.A
* $6 :! Impossible{.A = $5}
  - E.C
* $7 :! Impossible
  - D.E
  - E.D.B
* $8 :! Impossible{.B = $7}
  - E.D
* $9 :! Impossible{.E = $9}
  - C.C.A
  - C.C.A.E
* $10 :! Impossible{.A = $9}
  - C.C
* $11 :! Impossible{.C = $1, .D = $3}
  - A
* $12 :! Impossible{.C = $2, .D = $4}
```

```
  - B
* $13 :! Impossible{.A = $1, .B = $2, .C = $10, .E = $5}
  - C
* $14 :! Impossible{.A = $3, .B = $4, .E = $7}
  - D
* $15 :! Impossible{.C = $6, .D = $8}
  - E
* $0 :! Impossible{.A = $11, .B = $12, .C = $13, .D = $14, .E = $15}
  - Self
```

Only place where type checking isn't satisfied constructively is when you use `.Self` or `Self`.

```
interface E {
    let R:! Type;
}

interface A {
    let Q:! E;
    let P:! E;
    let Y:! A where .Q == Q and .P == Q;
}

interface B {
    let X:! A where .Y == .Self and .P == .Q;
}
```

The condition `let Y:! A where .Q == Q` is satisfied automatically for `let X:! A where .Y=.Self`. However `let Y:! A where ... and .P == Q` requires the `.P == .Q` restriction on `X`, otherwise it wouldn't pass type checking.

`Impossible` fails to type-check because, while we know that `$9` refers to itself when referred to within `C.C`, we don't know that the same applies recursively within `C.C.C`. Put another way, the value `$10` does not satisfy the type rule `$13` – the invented `$10` value does not type-check as an `Impossible.C` value – because we don't know that `$10.C.A == $10.C`.

zygoloid thinks we would see the same problem with other kinds of indirect self-reference:

```
interface Container {
  let Slice:! Container where .Slice == .Self;
  let Subseq:! Container where .Slice == Slice;
}
```

Here, we would not know that `Subseq.Slice.Slice == Subseq.Slice` and so would be unable to verify that the value we invent to represent `Subseq` is a `Container`.

Another undecidable word problem: https://screenshot.googleplex.com/BgjhEJstpmMPZiG

## 2021-09-16

- Attendees: chandlerc, zygoloid, jonmeow, josh11b
- Let's say we have an `i12` field of a packed struct called `y`, `a.y + b.y` should be an `i12` so it can be assigned to `c.y` without a cast
- Only `i8`, `i16`, etc. are legal outside of a packed struct declaration, inside a packed struct declaration, `i1`, `i2`, etc. and `u1`, `u2`, etc. are legal; if you really need an unusual size outside a packed struct declaration, use `Int(N)` or `Unsigned(N)`
- Only concerned about hardening `+` at a coarse granularity, not the expression level.
- Arithmetic between literals and a non-literal is a compile error if the literal doesn't fit in the non-literal's type.
- Only known literals can be converted to `Int(N)` or `Unsigned(N)`, not generic literals. Conversion operator is templated on input literal.
- Templates can have an `if` clause that can disable the function based on an arbitrary computation on the actual instantiated values of template parameters, and build constants like `IS_WIN64_BUILD`
- Will need something like adapters but with restricted casts: types that are produced after validation, facets that give capabilities like the read and write ends of a pipe being facets of the same value, private access, const access
- Want two kinds of `as`, following C++ having multiple casts; regular `as` would be for the safe things like implicit conversions, the other `as` would support truncation, etc.
- So two interfaces but no distinction between `As` and `ImplicitAs`, the distinction is between `As` and `UnsafeAs`
- May also have `TryAs` that implies `UnsafeAs`, but not all `UnsafeAs` come from `TryAs` e.g. casting away const. So `As` implies `TryAs` implies `UnsafeAs`, via blanket impl and possible requirement
- Question: have `fn F[U:! Type, T:! As(U)](x: T, y: U)` that calls `fn G[U:! Type, T:! TryAs(U)](x: T, y: U)`, is the blanket impl of `TryAs(U)` for things that implement `As(U)` enough for the call to `G` to succeed, or does `As` have to explicitly have a requirement on `TryAs`?
    - One concern with blanket impls is whether having one guarantees that you can instantiate. Question is whether instantiation can be blocked by a conflict with some other blanket impl that makes impl selection ambiguous.
- Do we expect inheritance to be used more narrowly than C++, for only its traditional use case of refinement of an abstract type?

## 2021-09-13

- Attendees: zygoloid, josh11b, jonmeow, chandlerc
- `Apple + Apple` should only look up implementations that are associated with `Apple` or `+`
    - Want to avoid a situation where `Banana` depends on `Apple` and introduces an implicit `Apple`->`Banana` conversion and an `Apple + Banana` operation and the ability to do `Apple + Apple` depends on whether `Banana` is imported
        - Three components:

        - `Banana` defines an implicit conversion from `Apple` to `Banana`
        - `Banana` defines an `Apple + Banana` operation
        - client code performs `Apple + Apple`
    - Combination of these three should be invalid, so one of these three should be invalid (or something like doing 1 and 2 together is invalid). But which is invalid?
        - We think we need `Apple` -> `Banana` for interop and modeling converting constructors
        - We think we need `Apple + Banana`
        - So `Apple + Apple` should not select (and maybe not even find) the `Apple + Banana` operation
- Concrete example:

```
// Represents a type that behaves like a tuple.
interface TupleLike { ... }
class Scalar { ... }
// Allow scalar * tuple.
impl [T:! TupleLike] Scalar as Mul(T) { ... }

class Vector(T:! Type, N:! i32) { ... }
impl [T:! Type, N:! i32] Vector(T, N) as TupleLike { ... }
impl [T:! Type] Vector(T, 1) as ImplicitAs(Scalar) { ... }

fn F(v1: Vector(i32, 1), v2: Vector(i32, 1)) {
  // OK?
  // Do we consider v1 -> Scalar then use Scalar * (Vector(i32, 1) as TupleLike)?
  let v3: Vector(i32, 1) = v1 * v2;
}
```

        - In C++, this would work if the `operator*` corresponding to the `Mul impl` were found by ADL in the associated namespaces and classes of `Vector(i32, 1)`.
        - In Carbon, we might be saying that this works if that `Mul impl` is defined with `Vector(i32, 1)`; this seems like the equivalent of ADL.
    - Use case: I have an `Coconut` that I want to be able to use anywhere I can use a `Durian`; in this situation we want there be an implicit conversion `Coconut`->`Durian` defined with `Coconut`, and want the compiler to consider that conversion anytime using values of type `Coconut`
        - `fn F(d: Durian)` should be callable with a `Coconut`
        - `Durian` implements `Smelly`, `fn G[T:! Smelly](d: T)`, can we call `G` with a `Coconut`? No: what if `Coconut` also implements an implicit conversion to `Ginkgo` which also implements `Smelly`?
    - Idea: could have `Compare(Int(N), Int(M))` for all `N` and `M` 1..256, but delegate to a finite set of compare functions for N == M == 2^k, where the caller will do an implicit conversion to the final type. Means that we allow generally interfaces to implemented with functions that don't have the exact right signature, but can accommodate the calling type via implicit conversions
    - Rule: always look up interface impls using the precise type

        - Never find `Apple + Banana` when looking for `Apple + Apple` regardless of where it is or if there's an implicit conversion
        - Can provide a generic `Apple + T`, for any `T` that converts to `Banana`, but that can only live with `Apple` not with `Banana` by the normal coherence rule
            - This is the same constraint as the ADL approach would provide, and the same behavior, but with implicit conversions made explicit
            - `impl [T:! ImplicitAs(Banana)] Apple as Addable(T)`
        - Should these be ambiguous for `Apple + Apple`? (Should we have `Apple` as `ImplicitAs(Apple)`?)
            - `impl [T:! ImplicitAs(Apple)] Apple as Addable(T)`
            - `impl [T:! ImplicitAs(Apple)] T as Addable(Apple)`
        - Other option is that there is no match for `Apple + Apple`
            - Prefer ambiguity, and a blanket impl `T as ImplicitAs(T)`.
            - Since that allows you to pass `Banana` to something that takes `ImplicitAs(Banana)`
    - Rule: you only ever apply an implicit conversion when you know the destination type
        - Implicit conversions allowed in pattern matching and overloads
- Aside: overloaded function
    - Idea:

```
overloads {
  same_specifiers fn SameName(...);
  same_specifiers fn SameName(...);
}
```

    - private vs. public differences: problematic in C++ because you can't resolve access until after overload resolution; can't do access checks as part of name lookup
        - doesn't bother ChandlerC as much
    - aside: issue with C++ using declarations with inheritance wanting to add a function with the same name as parent's overloaded function one of which is private one is public
    - would be nice
    - Concern with not repeating name is forward declaration matches out of line definition, josh11b likes:

```
overload SameName {
  fn (... signature 1...) -> ret 1;
  fn (... signature 2...) -> ret 2;
}

fn SameName(...signature 1...) -> ret 1 { ... }
fn SameName(...signature 2...) -> ret 2 { ... }
```

    - First match wins
    - If you want to override functions in an overload set in a derived class, have to build a covering overload set in the derived class

        - no synthesis of an overload set combining functions from the base and derived
    - Same deal in interfaces, impls have to cover all the overloads in the interface
    - josh11b: worried about evolution, trying to transition a virtual function to another with the same name but different signature
        - tying the public name to the virtual name is awkward
        - recommendation to use private virtual and separate public non-virtual interface
    - C++ design bug is the behavior of derived class hiding names in the base class
    - Another choice: derived class does not change overload set for a name defined in in its base class
    - `impl fn` syntax in derived class doesn't change lookup for that name in derived values, just lookup of names in the derived type, doesn't hide anything in the base class

```
// Can only define overloads with a forward declaration
fn SameName = overload
  (...signature 1...) -> ret 1,
  (...signature 2...) -> ret 2;
```

    - [zygoloid] For me overload syntax comes down to: how closely connected do we want overloads to be?
        - if these are implementing the same function / functionality, but just providing different interfaces, *and* they can reasonably all be declared together (rather than, say, declared with the type they operate on), then any of the above `overload` syntaxes seems mostly OK (but the extra indentation and disruption to the normal API flow seems somewhat undesirable)
        - otherwise – for example, if you're grouping the API by type, and have one overload per type, or if the overloads and their corresponding types aren't all defined in the same api file – this syntax seems problematic.
    - our choice of syntax will influence people's design choices; what designs do we want to encourage and discourage?
- Resume conversation about implicit conversions and impl lookup (motivated by binary operators)
    - [zygoloid] For a type `Apple`, I want to be able to write a single `impl` that provides all of these:
        - `impl [T:! ImplicitAs(Apple)] Apple as Addable(T)`
        - `impl [T:! ImplicitAs(Apple)] T as Addable(Apple)`
        - `impl Apple as Addable(Apple)`
    - … in which I only define the `Add` function once.
    - Idea: define a [structural?] interface:

```
// in the library
[structural] interface SymmetricAddable {
  fn Add(x: Self, y: Self) -> Self;
  impl [T:! ImplicitAs(Self)] Self as Addable(T) { alias Add =
```

```
SymmetricAddable.Add; }
  impl [T:! ImplicitAs(Self)] T as Addable(Self) { alias Add =
SymmetricAddable.Add; }
  impl Self as Addable(Self) { alias Add = SymmetricAddable.Add; }
}
// in Apple
impl Apple as SymmetricAddable {
  fn Add(a: Apple. b: Apple) -> Apple { ... }
}
```

        - [josh11b] What about?

```
fn AddApple(l: Apple, r: Apple) { ... }
impl Apple as Addable(Apple) {
  let Add = AddApple;
}
impl [T:! ImplicitAs(Apple)] Apple as Addable(T) {
  let Add = AddApple;
}
impl [T:! ImplicitAs(Apple)] T as Addable(Apple) {
  let Add = AddApple;
}
```

        - [josh11b] Could potentially use `where A != B` constraints to remove ambiguity
            - Also useful for defining `Compare(B, A)` in terms of `Compare(A, B)` but not when `A == B`.
- For interop with C++, need to support `Compare` interface for C++ classes that only implement `<`, etc. and not `<=>`; Carbon types will always use the default implementations of `<`, etc. in terms of a single `Compare` function equivalent to `<=>`.
    - Don't generally want Carbon types to have inconsistent sets of comparison operators, don't care about accommodating EDSLs using those operators
    - Rust macro_rules approach transforms tokens into Rust code, instead of using operator overloading to support EDSLs
    - So `<` always returns a Bool?
        - SIMD operators? SPMD instead
        - Boost lambda approach, expression templates? Nope
- Long talk about specialization
- Talk about #821: Values, variables, pointers, and references and comparison to C++
- Possible convention for returning values by reference, with the lifetime of the parameter with the shortest lifetime

## 2021-09-09

- Attendees: geoffromer, josh11b
- `Ptr` doesn't allow magic const casts that C++ pointers allow

```
int** p = ...;
int const * const* allowed = p;
int * const* allowed2 = p;
int const ** illegal = p;
```

    - This is about both a type-system thing (these const casts are safe), and a representation thing (the compiler can enforce that `int*` and `int const*` have the same bit representation)
    - Only known fix right now is to make a special `PtrVector<T>` that acts like `vector<Ptr<T>>` but is implemented in terms of `vector<T*>`, along with corresponding view types
- Would Carbon have the same issues?
    - #821 has a very different take on const support
    - no way to have a per-instance `let` members in classes
    - Are we going to have the same situation as Java that has a proliferation of types to represent immutable and mutable versions? And proactive copying of data passed to an API to ensure that your copy is not changed.
-

## 2021-09-07

- Attendees: chandlerc, zygoloid, dhollman, geoffromer, josh11b
- [josh11b] anything to grease the wheels of figuring out keywords / spellings?
    - [chandlerc] Could use the painter approach more often
    - [chandlerc] The motivating example the keyword choice is load bearing: not just choosing between things that are about equally acceptable
        - People have significant concerns about some choices
    - [chandlerc] No changes to the design are really bubbling to the surface as we discussed keyword options. Concern was whether the difficulty choosing the name meant that the design needs to be fixed, but now that we've focused the facet as just a solution for ABCs can now just use a placeholder if necessary so we can move on.
    - Random bystander thought of these as "partial base classes", may just use `partial` since it has relatively few problems even if not much love
    - Kate liked "we don't have constructors", motivated this solution
    - [zygoloid] Making this about ABCs, decoupled this from `.base = ...` in the derived constructor
    - [chandlerc] Concern was we were going to be talking about these a lot before we restricted to ABCs, so this was an important piece of terminology not just a keyword
- [josh11b] thoughts about https://matklad.github.io/2021/09/04/fast-rust-builds.html?
    - [chandlerc] Interesting when people dig in and analyze things; particularly the strategy for generating code for generics
        - [chandlerc] the generated code for a generic is not done at the granularity or location expected; it is not the crate that defines or uses the generic, it is more like C++ template instantiation in that it is per use

        - [chandlerc] there are traits that are just generic around a type conversion, after which the function is not generic; there is a now a crate that splits the function into a generic conversion and a non-generic body
        - [chandlerc] Rust proc macros traditionally runs in a symbolic interpreter, slow for running a Rust parser, can now compile the proc macro to WASM
        - D runs a JIT, Circle allows you to pull arbitrary system library in process for running meta code -- e.g. curl to download a missing dependency
        - [chandlerc] Carbon should look for cut points after which a function is no longer generic; `momo` supports three kinds of cut points, `as_ref`, `into`, and one other. Maybe implicit conversions are happening in the generic instead of being done in the caller.
        - [chandlerc] Large trade-off in how you generate code for generics between putting code into the generic vs. putting code into the caller.
        - Is Rust doing any linkage based elimination of duplicate instantiations? Or doing a lot of inlining in the caller so it can't?
        - 5x template deduplication typical for C++
        - [chandlerc] Rust should look at the tricks C++ uses to scale templates; limits without language or library changes
        - [geoffromer] extern templates?
            - [chandlerc] used infrequently but with high impact, e.g. in standard library, protocol buffer compiler
            - [zygoloid] `inline`'s interactions with extern templates -- tried still generating an instantiation for each caller for inlining, but it gets thrown away if not used -- but massive space regression
            - [chandlerc] Not duplicating code is not always a win due to less inlining; strategy can work in debug builds
            - In C++, no way make use of an analysis of a function that determines if it will always or never
- Compilation model?
    - Do we have one compiler invocation per file, or one per library? Per file.
    - Do we have a step that packages the components of a library together?
    - [chandlerc] Rationale: easiest to teach to humans and build systems mechanism for parallelization is at the file level granularity
    - [zygoloid] Want to reduce rebuilding if something distant changes; can we copy from dependencies to allow us to detect changes that don't matter to transitive deps
        - Goal is the compiled form of B that depends on A, includes all the information for all Cs that depend on B
        - Can we measure what percentage of A that is exposed by B as heuristic to decide whether to copy A into B?
        - Concern is greatly increasing size due to duplication-- is every file going to duplicate some common libraries like `string`?
    - Alternative has scalability concerns
        - One concern is staleness information
        - One alternative is hashing your dependency, to prune downstream changes

        - Question:
          C depends on B depends on A
          B is not affected by changes to A, and doesn't change B's interface
          How do we decide that we don't need to rebuild C?
        - The input files to C's compilation is different than the set of files checked to decide if C needs to be rebuilt
        - If we had a content-addressable-database of build artifacts, could re-export everything B depend on in A just by exporting hashes of the things used from A
        - A could precompute hashes for a fine granularity of its output
            - Could use the Merkle tree optimization then
        - Could use duplication to verify that the hash is correct
        - Deduplication falls out of using hashes
        - Can optimize the fetch of C's needs from A using the hashes; but worry that the size might be smaller than hash
        - Use hashes instead of mangled names
        - Potential optimizations (particularly for local builds) to avoid recompiling C based on what it used from B being listed in C's output
            - distributed builds are probably more concerned about transfer costs, don't necessarily have the previous output available
            - local builds, the tests are for things that are hot in caches

## 2021-08-30

- Attendees: jonmeow, josh11b, chandlerc
- Discussed options in Extensible classes
    - Leaning toward the more C++ options, possibly just C++ with the modification that you have to say `unsafe_delete` to delete an `Extensible*` without a virtual destructor.
    - Could revisit when we have less time pressure and more data.
- #741 constructors
    - Three most viable options:
        - `factory`/`construct`
        - regular functions with no additional safety beyond C++, but need something for abstract classes
            - possibly a keyword both as the return type and and the call site when constructing an abstract base class
            - can call virtual functions on the base object before it is returned, get the vtable from the base object
        - regular functions with a weird type for extra safety
    - Three possible goals:
        - regular functions instead of special constructors
        - no performance cost from reinit vptr
        - safety advantage of virtual calls not being possible when they aren't as meaningful

    - Deciding that first goal is the most important
        - expert users that very much care about performance can avoid re-initing by using the keyword
        - still will prevent virtual calls in abstract base classes, eliminating pure virtual function calls
    - Still some performance concerns if the class is inheriting from lots of interfaces, but that is expected to be rare
    - Maybe people who were very safety focussed would provide two versions of the constructor functions
    - Would good to be have a default way of converting certain kinds of argument lists to a type -- like list of vector elements to a vector
        - model that as a conversion
    - Rule: wherever in C++ you would use an explicit constructor, use a named factory function instead; wherever in C++ you would use an implicit constructor, use a conversion function
        - Carbon will use interfaces to support making conversions bidirectional more easily
    - Could call this the `concrete` version of an abstract class, wasn't a big hit
    - Would we use this variant type for both constructors and destructors?
- Destructors?
    - classes all need them, though a default can sometimes work
    - no arguments, but can be virtual or not
    - some classes want customized code that is run when they are deallocated, which might have parameters passed to it
    - in contrast, being destroyed is implicit, no arguments
    - Rust called this "drop" but it was a mistake to be different, should call them "destructors"
    - Are destructors implemented via an interface, or they are an intrinsic part of the class
    - implementing an interface is a bit verbose
        - maybe we want a way to make a special case for single-function interface?
        - better would be to have a shorthand for implementing an interface with a single function -- maybe `impl fn MyInterface.MemberFunction... { ... }`
    - private destructors? would we need to express the capability to destroy as a generic constraint, which leans towards interfaces
    - what behavior do you get if you don't implement the interface? private destructor or default destructor
    - what about getting the default destructor when you want to make it virtual?
    - private destructors are rare, comparable to being unsized
        - classes should implement these traits automatically, have an opt-out
        - could have some syntactic sugar for the obvious customizations when you want to opt-out
    - having a special keyword for the destructor doesn't bother Chandler

- previously said we don't want to support destructors failing, have to deal with possible failures before destruction
- what if we allow you to have a return type for your destructor, but when it isn't void the destructor cannot be implicitly called
  - requires you to explicitly call the destructor (and handle its return type potentially) prior to the end of lifetime
  - doing so puts the object into an unformed state (and so you have to have an unformed state)
    - this is a bit recursive -- if you have an unformed state we're allowed to run the destructor on it?
    - require the unformed state to not be one that will fail?
    - Or not unformed, could just end the lifetime of the object
  - These destructors would have to be named, you could have two different ways of destroying the object
    - Really, these are an alternative to a destructor
    - `class MyClass { fn MyDestroyer[destroy me: Self](...) -> ... { ... } }`
  - Keep this in mind, but don't need to implement it now
  - Part of the error-handling story, when we tackle that
- Do we want to allow calling virtual functions in destructors at all?
  - ability to call virtual functions in destructors creates a data race in the C++ language that can't be avoided, since write to vptr is unsynchronized
  - could potentially avoid this by requiring destructors with synchronizing calls to be final, since we could avoid the initial vtable assignment
  - TSAN works around this by detecting whether the vtable assignment is to the same value when there is synchronization
  - Not a lot of value in calling virtual functions when you won't be getting the derived version
    - Maybe calling a base class function that turns around and calls a virtual method?
    - Not that useful, but preventing it might be hard
  - Could just null the vptr on the way in, but it will break synchronization in a destructor
    - good case for final, since deriving from a class that does synchronization is unlikely to be sensible
  - Seems most sensible only in most derived type, so tempting to only allow it in final classes
    - but still may want to put common destructor code, including synchronization, in a base class helper function, which may turn around and call a virtual function
  - Maybe we just zero the vptr when we get to an abstract base class' destructor, a la a partial type, and fix the race condition by fixing the ABI
    - it isn't the destructor that sets the vptr for itself, the destructor will set it to the base class' at the end
    - don't have to worry about C++ ABI issues around virtual base classes

        - Syntax: `[private] [virtual] destructor(; | { ... })`
            - there will be some other thing going on with interfaces
        - Do we want to support undestroyable things? Eternal objects that are never destroyed? Probably not worth spending effort on this now
- private interface implementations?
    - context sensitivity vs. access control facets
    - context sensitivity awkward when forwarding to an interface
    - for now could just say: if you have a public destructor, you automatically implement the destroy interface
    - could just restrict construction instead of destruction
    - case where you construct something and hand it off to some other function that should not delete
        - really that is about not transferring ownership when you pass a pointer
        - want something more restricted than a C++ pointer, possibly by changing the ownership semantics of pointers
        - a difference between an owning and unowning pointer, like C++'s `unique_ptr`
        - default pointer is non-owning
    - For now, no access control on destructors; syntax: `[virtual] destructor(; | { ... })`
- Do we want to insist that you introduce a class with abstract functions using the abstract keyword?
    - Stronger than in the current proposal
    - Puts very important information up front for readers
    - Changing a type from extensible to abstract is a radical change, will probably update the comment, rename it, etc.
    - Don't need to say "protected destructor", just make it abstract
- If base class defines the destructor as `virtual`:
    - need either `virtual` or `impl` on destructor in derived? or use the same rule as methods?
    - simpler to match methods -- even though it makes evolution a little trickier, and destructors don't have the same overload-matching issues
- We will allow you to construct extensible objects with non-virtual destructors
- Will need to expand #652 to cover all the extensible object issues
- #780 on constraints
    - Chandler prefers "whole expression constraint intersections" over "per-interface inputs"
    - Want to understand about a type, can find everything about it by looking at the declaration of that type and recursing into things that are mentioned
    - Chandler's biggest problem with constraining inputs is syntactic; the passing in syntax itself, too similar to parameterizing
        - maybe pretty rare to do both
        - likes both `where` syntaxes, thinks they are clear
    - leaning towards just `where` type expression
        - restricted
        - affects api

        - allow `.X` on either side of the `==`
        - can also have arbitrary type expressions
        - for recursive constraints: `.Self` refers to `Self` in the interface's scope
        - implies `MyInterface` actually has a member named `Self` that is created automatically -- as much as any other associated type of `MyInterface`
        - later may need to use a separate keyword on declarations (`requires`?)
        - want the `where` to only affect the API of the type it is modifying,

```
fn F[T:! Type, U:! Container where .Element is Comparable, .Element = T](u: U);
```

gets rewritten to

```
fn F[_1:! Comparable, let T:! Type = _1, U:! Container{.Element = _1}](u: U);
```

    - Question: how does this interact with templates?

## 2021-08-26

- Attendees: chandlerc, josh11b, zygoloid, mconst
- Non-abstract extendable types
    - We have some operations that we either never want to perform on these extendable types, like slicing style assignments, or only want to perform under certain criteria like for local variables.
    - Natural to separate the types: sometimes you have a type that has the constraints that allow operations, sometimes you have a different type without those restrictions but can't perform all the operations.
    - Concerns are:
        - C++ legacy (not the primary concern)
        - needing to use 2 different types
        - awkward code changes as use cases evolve
    - Having more types is expensive for humans, cognitively
    - How would we do this without types? We should separate out these operations into distinct operations.
        - For example, separate operations for static and dynamic destruction
        - Rather than having two different types that have a difference of whether they support a single operation.
        -
    - Want to preserve the out that if extendable types have limitations you can always switch to just using final types
    - [zygoloid] maybe these are type-of-types... and when we use an extensible type as a type, it is an *exact* type. When we use it as a pointer, it is type-or-derived.
        - Basically `Base as Type` is an exact type, and `Base as InexactType` is what `Base*` uses to enable pointing to derived types.
        - <lots of discussion seeming to like this>
        - one concern is conversions...
        - Basically, this gives us two pointer types -- which we avoided in the past because it risks an explosion of generics parameterized on pointers.

somewhat a question of -- can we minimize that / avoid that enough to be successful?
    - [mconst] pushing back towards making the distinction between the class type rather than the pointer type
        - maybe we can make this look similar enough to the type-of-type approach?
        - but keep the difference in the class type rather than the pointer type
        - seems nicely promising in most ways
        - but postfix star on types isn't injective...

```
let T:! Type = Base;
var v: T*;
```

        - but now `T*` always means T-or-derived for non-final T. tolerable.
        - but how can we define things on extensible classes that are only available when we have the exact type?
        - unclear we care -- opinions differ...
        -
- [speaker] Comments
    - More comments

## 2021-08-23

- Attendees: chandlerc, dhollman, zygoloid, jonmeow, josh11b
- Decision: `alias X = Y`
- Discussion on virtual, override, pure virtual
    - We don't like the `= 0` suffix (or `= pure`), since it is in place of an implementation but we wouldn't put it out of line like an implementation. So should put it wherever we are saying `virtual`, etc.
    - Want to match C++ keywords when semantics match, argues for `virtual` over other choices
    - Concern: evolution from a base class to an interface, which you can still derive from
        - overriding a virtual function from a direct or indirect base class compared to implementing a method in an interface that you directly or indirectly derive from
        - conclusion: the intrusive dyn-ptr case should use the inheritance syntax; deriving from an interface gives you pure-virtual methods
    - What does `Self` mean? In an interface it means the type implementing the interface, in a virtual method, is it the lexically enclosing class or the most derived class? To match C++, we probably want the lexically enclosing class.
    - When we derive from an interface, we first convert the interface into a base class (an "interface class"), the `Self` in the methods of that interface base class refers to the interface base class
    - Do we have a use case for `final` methods?
    - Case: defining a function as pure virtual and implementing it

        - e.g. the destructor, to make the class abstract even though every method is implemented
        - also to provide an implementation in the base class but still require descendants to implement it -- better to just use a protected helper method
    - Maybe `virtual` is the default in interfaces and `final` is the default in classes
        - Problem: interfaces may use static dispatch, mismatch with `virtual`
    - Josh demands a decision: `abstract` wins over `pure virtual`
        - Still talk about "abstract virtual functions", but "abstract" on its own serves as clear indicator, unlike "pure" without "virtual"
    - Three use cases
        - I'm providing an extension point, you must implement
        - I'm providing an implementation, you may override
        - I'm providing an extension point, but there is a default
        - Last two use cases are different but both get `virtual`
        - Last use case might be replaced by a helper function and a pure virtual, needs a cheap way of saying "implement this with that"
    - Question: use `override` in the same position as `virtual`
        - concern: overriding an abstract function isn't really "overriding"
        - could also use `impl fn ...`, consistent with implementing interfaces
    - josh11b and zygoloid agree we want mandatory syntax; chandlerc points out that we can instead forbid overloading in this case
    - `impl fn` would never change the names in the class
    - Question: would you allow `impl fn` out of line, without a declaration in the class?
        - No: problem if it is abstract in the base, does an intermediate class implement it so a final class doesn't?
    - Do out-of-line definitions need to carry `private`? Similar to the `impl` question.
    - Decided : `impl fn` instead of `override fn`
- `ctor`, `under_construction`, `partial`, `construction`
    - Make it an error if you use the keyword with a final class
    - Argued against `base` again
    - josh11b: `construction` by fiat, will likely need to be revisited since we clearly don't have agreement
- "Three thumbs up" or "lead's judgement"? Leaning toward lead's judgement
- Do we need `final` for methods?
    - Can be dropped without breaking code
    - Get similar performance benefits from profile guided optimizations and devirtualization
    - Could imagine using `final` to mean "no shadow"
    - Wait and see if it is useful for Carbon programmers, won't include `final` for now
- Discussed upcoming leads question on constraints
    - Fixed some bugs in restricted `where` approach
    - Considered constraint that expand into cyclic references
    - Do we want to allow forward declaration of interfaces, to allow cyclic reference between them?

    - Worked an extended example using graphs, edges, and vertices to work out ways to work around limitations
    - Possibly `where` clauses change type but not API, a bit like external impl
    - Do constraints reach into interfaces to change types or just allow casts?

## 2021-08-16

- Attendees: chandlerc, josh11b, mconst
- Talking about Carbon: inheritance
    - default of "final" for classes is scary but likely doable; definitely needs a rationale
- [chandlerc] I have terrible ideas about placement of virtual keywords

```
base class MyBaseClass {
  virtual {
    fn Overridable[me: Self]() -> i32;
    fn MyPureMethod[me: Self]() -> f32;
  }

  fn Overridable[me: Self]() -> i32 { return 7; }
}
```

- [chandlerc] This virtual block works the same way as an interface
- Could we make default methods of interfaces work like virtual method implementations?
- Restrictions here would be the same as "object-safe" restrictions on interfaces
- Could use `final` in an interface to represent non-virtual methods in an abstract base class
- Block approach helps optimize for the two different readers of the class
- Another place where block approach can help is for modeling `override`:

```
base class MiddleDerived extends MyBaseClass {
  // Lots of introducers available, such as override MyBaseClass etc.
  impl as ??? MyBaseClass {
    virtual fn Overridable[me: Self]() -> i32;
    virtual fn MyPureMethod[me: Self]() -> f32;
  }

  virtual fn NewOverridable[me: Self]() -> i32;
}
```

- Question: assigning from a struct value that does not specify all of the fields to a nominal class. See context.
    - Breaks assignment using a blanket implicit conversion from struct values.
    - Is this actually a foot gun? Maybe a lint rule? Maybe a lint rule only later if it proves to be an issue?
    - Do we actually want to allow implicit conversion from struct type values to nominal classes in assignment at all?

    - In C++ variable initialization is very different from passing something to a function. Can disable implicit conversion in C++, by deleting a templated conversion. Two categories of implicit conversion in C++, a conversion operator can be marked `explicit` -- which just disables some conversions in some contexts.
- Question: should we allow unqualified name lookup, or should all names in methods be qualified?

## Archive

- Jan-Aug 2021