GOAL :

- Introduce a friend to the PL.
- Incrementally develop concepts
  - dependency order of a feature
    ⇓
    impl. order " the .
- Document whole feature-set [I keep forgetting]

TICKETS

Things to implement.

① —— ⑭ in Black ink

MIND MAP

BY SYNTAX

- patterns
- expressions
- statements : const, class
  ⑦ ⑧
- modules
- source files

BY RUNTIME (AOT vs. RT)

- const / non-const (const)
- non-const immut / non-const mut (&)

⟹

BY TYPING

- Atomic Types : S_, U_, F_
- Aggregate Types : { ... }, enum
- Function Types : (T) → U
- Template Types : <T Any, u S32> → <T Any>
  could be <v Any> for value (14)
- Pointers : ^T
- Mutables : &T —— 6
  e.g.: ArrayLike class : T :: <...> → {...}
- ⑧ Classes : sets of types
  class C (T Any) <...> {
  ⑨ type ~~req~~ T.x :: F32 ;
  ⑩ val ~~chk~~ (v T) = (v.x == 144)
  } ;

↳ chk imposes a condition on ALL values of a type.

replace with 'type' & 'val'; ~~'test'~~ ~~'check'~~

↳ For templates, —— ⑪ (explicit call?)

① Check spec indep:
   if specced constraints bad, def^n error.
② At tcall [explicit / implicit], B
   check class instanceship
   At vcall,
   check type instanceship
   Mismatch? Call error.
③ to emit, just emit per monomorphic permutation

Templates ⇓ class OK + in class OK

⑫ : enable <> args (explicit)
↓
⑦ : const stmt
↓
⑬ : enable [] args (implicit)
↓
⑭ : enable (v Cls) implicit template param

MISC :

- '.' for static + dyn lookup
  ① - .(?) for indexing
  cP enum ⓪ - T.id ⇒ type of field
- ':' for typing ops :
  ② - '::' for eq,
  ③ - ':<' for ~~subtype~~ sup? fewer items in struct?
    U8 :< U16 ∵ conv sub → sup
    for compound, match eq size + fieldwise :<
    {x U8} :< {x U16}
    {y U8} ~~:<~~ {x U16}
    name ×
    req T >: {
      x S32, y S32
    }
    {x U8} :< {x U8, y U8}
    NOTE: order independent
    only top-level fields consider 'embed'

Types feature

{U8} :< ?
No : tuple != struct

④ - reintroduce → as *(). , lambda "=>" ⑤
  ↳ clearer typing ::
  T.x != T→x
- let &x = 42;
  let (&x, y) = (42, 52)
  ⑥

* Dynamic bdd allocation (WIP : allocators, library ?)

① Fixed bound
'dim' stmt creates a REGION with opt (non-const) max-size in B.
e.g. dim positions Vec3 (n) @ r? How do we know? type of region arg?
Each region is ...
① disposed of on return (dropped) UNLESS RETURNED
② contained in heap or another region
   no "region"
③ used to allocate with 'new' stackwise
From CFG, compiler can:
⓪ trace each 'new' to a parent 'dim' / else error!
① check / compute bound for dim to compare
② Dynamic bound
Get free data & use to set resize steps < total upper bound

may be a value!

tcall ⇒ template call
vcall ⇒ runtime call / value call
vtcall

⑮ shorten structs

* IDEA:
  - use an allocator for numerous objects
  - tie allocators to CFG frames

① Change ':' to '.' for module lookup (no → possible)
  - in typer, T_MODULE known during postfix visit
  - early overloading: when overload determinable from scopes alone (no unknowns)
  - else, '.' assumes LHS struct

⑥ Typespec . ID ⇒ type of field (and →)
  - enum Opt <T Any> {
      some T ;
      none ; # ⇒ unit, ; vs. , with groupings
    } ;
    Opt.some value ⇝ cast expr syntactically — special enum case!
    enum Color <T Any> {
      rgba {r, g, b, a T} ;
      hsva {h, s, v, a T} ;
    } ;
    Color < T >.rgba.r :: T
    (enum field type; struct field type)

WHAT OF MONOMORPHIC
- structs — (-1)
- enums — (-2)

② '::' type eq binop
③ ':<' and '>:' type binops
④ '→' ptr. (like C/C++)
⑤ Lambda ' |pattern => body'
⑥ '&' in |pattern items
  + &T typing
  + &T typechecking
    - ^ : &T → ^&T
    - ^ : T → ^T (still take ptr to lit, not 'const' !!)
    - set : ^&T = T
    - call : &T → T but T ↛ &T
      ^&T → ^T but ^T ↛ ^&T
      &(^T) → (^T) but (^T) ↛ &(^T)
      (some case; not confusing &/x accids. switched.)
    - dot : (&T).x → &Y where T :< {x F32, y F32} (? switch?)
    - struct/enum: error if field is mut: mutability set by owner
      ↳ warning?

⑦ const stmt
⑧ class stmt
⑨ type
⑩ val
⑪ <> tcall defns
⑫ <> tcalls
⑬ [] tcall defns + tcalls (explicit but inferred)

IMP! scope chaining / multiple

vs.

(14) (v Cls) implicit templates ⟶ ✗ : use `v Any<Cls>`

Typer Plan

- Types are just objects — not directly '==' comparable
- apply_rule ⟶ { ok / Fail / Deferred } — diff 'rules' for diff syntactical cases
  * rules ⟶ constraints : sup/sub, is_kind, convert_to (vcall), cast_to (cast<T>(v)), has_field
  * diff kinds of constraints ⇒ diff kinds of types
  * 'hypothesis' : free vars only
    - possible kinds bitset (so kinds Po2)
    - super & sub record vectors ⟶ (sup + constraint / rule)
    - fields

    ↳ constraint → rule

    rule → loc / why
  * on apply_rule,
    1. Rule (type) ⟶ Constraints ...
    2. for each constraint c,

       if c is known (not dep on free var)

       OK / Fail

       else

       update FreeVar with constraints
       - any conflicts ? FAIL
       - any changes ? DEFER after updating
       - else OK (confirm, no new info)

Constraint
- building block of rules
- tracks parent rule

KINDS :

1. Unit
2. Int
3. Float
4. Struct < (13) Tuples, (14) Typeclasses
5. Enum
6. VFunc
7. TFunc : maps FreeVars to sub packs
8. Any : emits as enum, cast from any type[x].kind == type.Kind.enum

9. Module
10. FreeVar : just a free variable, to be subbed later (cf metavar) (cf TFunc)

11. Type
   - reqd for Module field, tfunc

   t: Type ⇔ t is a type
   * classes are not types.
   (?) consider A<Cls> in specs ?
   (?) consider no TID / vid

   ∴ x a<Class>
   ≢
   x Class (not allowed)
   so
   x class ?

   so

   (*) — using so 'a' in usable name-space ?
   else type.a ✗

   so

12. Class

RULES :

1. ModRule (M, (string, T)[]) ↷ spans?

   kind (M, MODULE)

   for n, t in T

   M has field (n, t) ⟶ unnamed field ⇕ tuple

2. Let Rule
3. Def Rule
   ⋮

} of Typer ' ?

(A)

* IDEA
  - change 'class' to ~~'template'~~ typeclass
  - add OO-style classes : class, { using, dim, def }
  - add 'using' statement to import symbols.

value classes
- Ints
- Floats
- Strings
- Tuples
- Arrays
- Structs
- Enums
- Objects

≠ (B)

* uses '->' access operator for methods
* use '.*' for ptr access

  e.g. a.*b.*c

(15) Change TIO/VIO semantics : don't need separate !

(16) Add 'All' class : set of all types

T All ⇒ T is any type

(17) Add 'objects' as a new obj kind

- support methods, overriding

* add '=>' operator for method access
* add ∅-arg methods

:: x => y

could be useful, is v. explicit

* note x.y still used for data members

(18) * using y T ⇒ import / embed

* using a class includes methods w/ overloading / shadowing

```
class B {
    hi (x T) → T {
        x
    }
};

class C {
    using b B;
    member y X;
    method hi (x T) → T {
        hello (x)
    };
    abstract hello (x T) → T;
};
```

→ MUST be subtype of shadowed base

(19) Multi-ID variation field w/ shared fs

* Subclassing , Subtyping , Conversion

① A, B types

A subtype B

⇒ Bs substited by As

e.g. : 132 :< 164

⇒ B from A in call OK

i.e. can pass 132 for 164 w/o losing info

e.g. : {x, y u, ... T[]} :< {x, y T}

⇔ T :< u

i.e. can pass {x, y u, ... T[]}

as {x, y T}

⇔ T :< u

e.g. I32 :< Float64

i.e can pass i32 as f64

* A sub B

⇔ if f(B) → x, then

f(A) → x' [s.t. x' sub x]

* Rules :

① Int & Float :

$I8 :< I16 :< I32 :< I64$

$U8 :< U16 :< U32 :< U64$

$F32 :< F64$

② Structs :

$\{ a_1 \dots a_m \; T_1 \dots T_m \} \quad$ → all names unique

$\vdots$

$\{ b_1 \dots b_n \; u_1 \dots u_n \}$

$\Uparrow$

① $m \geq n$ (sub bigger)

② $\forall \; 1 \leq i \leq m,$

$\exists \; 1 \leq j \leq n$ s.t.

$\text{name}(b_j) = \text{name}(a_i)$

$T_i :< U_i$

$\{ x \; I32 \}$

$\vdots$

$\{ x, y \; I64 \}$

③ Enum

Each enum accepts a struct !

Follow struct typing rules.

④ VFunc

$A \dots \to B \; :< \; C \dots \to D$

$\Uparrow$ (A

$\text{len}(A) = \text{len}(C)$ ∧ $\forall i \; A_i :< C_i$

⑤ TFunc

see VFunc, Class

* ⊕ Subclass

$C :< D$

$\Leftrightarrow \; \forall \; T \in C, \; T \in D$

$\Leftrightarrow \; C \subseteq D$

for tcalls, not vcalls

* consider sep. subclass / subset op ?

* Idea for classes

① public/private labels : mandatory before ANY methods, [space before for 'friends']

② no shadowing in public

i.e. cannot embed class st. public fields in conflict.

```
class X {
public:
    x F32;
};
class Y {
public:
    extend base X;
    x F32;
};
```

↳ error ! public field "x" overridden