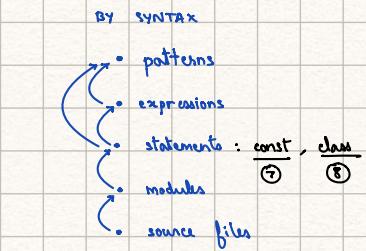


GOAL:

- introduce or friend to the PL.
- incrementally develop concepts
 - dependency order of a feature
 - ↓
 - impl. order "the"
- Document whole feature-set [I keep forgetting]

MIND MAP



- BY RUNTIME (AST vs. RT)
- const / non-const (const)
 - non-const immut / non-const mut (2)



BY TYPING

- Atomic Types : S-, U-, F-
- Aggregate Types : { ... }, enum
- Function Types : (T) → U
- Template Types : <T Any, n s32> → <T Any>
can be <r Any> for value (14)
- Pointers : ^T → 6
- Mutables: &T → 6
- Classes: set of types
class C (T Any) { ... }
- type tag T.x = F32;
- val etc (v T) = (v.x == 11h)

e.g.: ArrayLike darr: T → { ... } → { ... }

} replace with
'type' &
'tag'
'func'
'struct'
'val'

MISC:

- ① - `::` for static + dyn lookup
- ② - `()` for indexing
- ③ enum - Type of field
- ④ - `::` for typing ops:
- ⑤ - `::` for eq,
- ⑥ - `::` for subtype ^{sup?} _{func items in struct?}
- ⑦ - `::` for compound, match eq size + fieldwise is
`{x w6} :: {x w6}`
- ⑧ - `::` for compound, match eq size + fieldwise is
`{x w6} < {x w6, y w6}`
- ⑨ - `::` for compound, match eq size + fieldwise is
`{x w6} < {x w6, y w6}`
- ⑩ - `::` for compound, match eq size + fieldwise is
`{x w6} < {x w6, y w6}`
- ⑪ - reintroduce → as *(), lambda "=>"
- ⑫ - let &x = 42;
let (x, y) = (42, 52)

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

① Fixed bound

'dim' stmt creates a REGION with opt (non-const) max-size in B.

e.g. dim positions Vecs (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

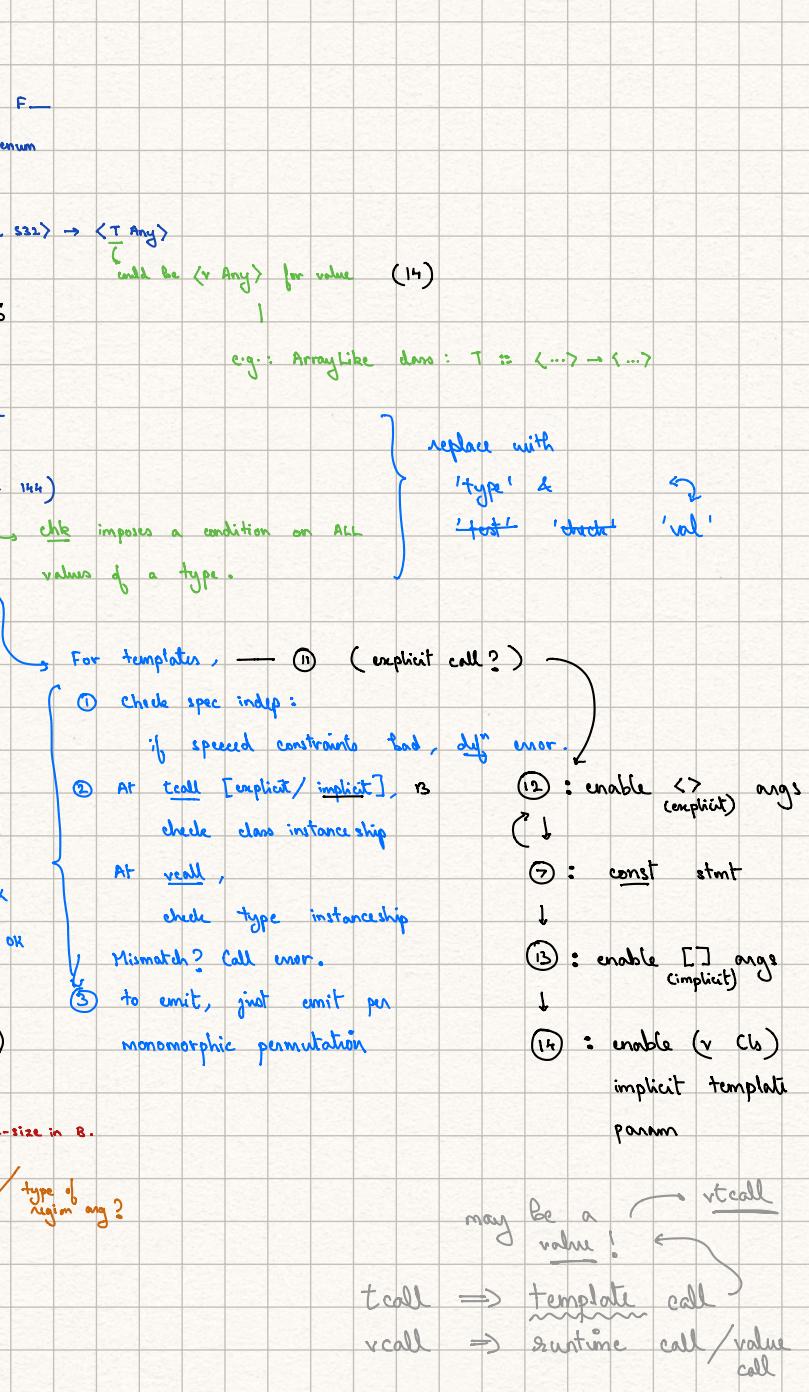
From C++, compiler can:

- ① trace each 'new' to a parent 'dim' / else error!
 - ② check/compute bound for dim to compare
 - ③ Dynamic bound
- Cut free data & use to set maxsize stns < total upper bound

TICKETS

Things to implement.

① — ⑭ in
black ink



⑯ shorten structs

* IDEA:

- use an allocator for numerous objects
- tie allocators to CFC frames

① Change `::` to `.` for module lookup (no → possible)

- in types, T_MODULE known during postfix visit
- early overloading: when overload determinable from scopes alone (no unknowns)
- else, `.` assumes LHS struct

② Typespec `T` → 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;};
    hsla {h, s, v, a : T;};
};
```

`Color < T >.rgba.r :: T`

③ `::` type eq binop

④ `:<` and `>:` type binops

⑤ `->` pth. (like C/C++)

⑥ `?` in lpattern items

+ &T typing

+ &T typechecking

- ^ : &T → ^&T

- ^ : T → ^T (still take pth to lit, not 'const' !!)

- set : ^&T = T

- call : &T → T but T ↳ &T

 ^&T → ^T but ^T ↳ ^&T

 &(T) → (^T) but (^T) → &(T)

] same case; not confusing E/I acciden. switched.

- dot : (&T).x → &y where T :< {x : F32, y : F32}
 ? match?

- struct/enum: error if field is mut: mutability set by owner
 ↳ warning?

⑦ const stmt

⑧ class stmt

⑨ type

⑩ val

{ ⑪ <> tcall def's

⑫ <> tcalls

⑬ [] tcall def's + tcalls (explicit but inferred)

↳ imp! scope chaining / multiple

WHAT OF MONOMORPHIC

- structs — (1)
- enums — (2)

vs.

⑭ (v Cls) implicit templates \longrightarrow X : use ' v Any<cls>'

Type Plan

- Types are just objects — not directly '==' comparable
- apply-rule $\rightarrow \{ \text{ok/Fail/Deferred} \}$ — diff 'rules' for diff syntactical cases
 - * rules \rightarrow constraints : sup/sub, is-kind, convert-to, cast-to, has-field
 - * diff kinds of constraints \Rightarrow diff kinds of types
 - * 'hypothesis': free vars only
 - possible kinds: bitset (so kinds P02)
 - super & sub record vectors \rightarrow (sup + constraint/rule)
 - fields
- * on apply-rule,
 - ① Rule (type) \rightarrow Constraints ...
 - ② for each constraint c_s ,
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)

RULES:
 ① ModRule ($M, \{ \text{string}, T \}[]$)
 kind(M , module)

for n, t in T
 M has field (n, t)

② Let Rule
 ③ Pd Rule
 :
 } of Type 1?

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

value classes

Ints

Floots

Strings

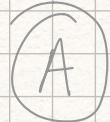
Tuples

Arrays

Structs

Enums

Objects



Kinds:

① Unit

② Int

③ Float

④ Struct

⑤ Tuples

⑥ Enum

⑦ Typeclasses

⑧ VFunc

⑨ TFunc : maps freevars to sub-pieces

⑩ Any : emits an enum, cast from any type [sc].kind == type::kind::enum

⑪ Module

⑫ FreeVar : just a free variable,
to be subbed later
(or mutator)
(or Tfunc)

⑬ Type

— need for module field,
tfunc

t: Type \Leftrightarrow t is a type

* classes are not types.

- ③ consider $A<cls>$ in specs?
 ④ consider no TID/vid
 \therefore no $a<\text{class}>$
 no Class (not allowed)
 so
 no class ?

⑤ using so 'a'
in readable name-space?
else type-a x

⑥ Class

- * uses ' \rightarrow ' access operator for methods
 * use ' $\cdot\ast$ ' for ptr access
 e.g. $a\cdot\ast b\cdot\ast c$

⑯ Change TIO/vio semantics : don't need separate!

⑰ Add 'All' class : set of all types

 T All \Rightarrow T is any type

→ ⑯ Add 'objects' as a new class kind

- support methods, overriding

* add ' \sim ' operator for method access

* add \otimes -arg methods

$\because x \sim y$

 could be useful, is v. explicit

* note $x \sim y$ still used for data members

⑰ * using y T \Rightarrow import/embed

* using a class includes methods

w/ overloading/shadowing

⑲ Multi-D return field w/ shared ts

```
class B {  
    hi (x T)  $\rightarrow$  T  
    x  
};  
};;
```

```
class C {  
    using B;  
    member y x;  
    method hi (x T)  $\rightarrow$  T  
        hello (x)  
    };  
    abstract hello (x T)  $\rightarrow$  T;  
};;
```

MUST be subtype
of shadowed base

* Subclassing \rightarrow Subtyping \rightarrow Conversion

① A, B types

A subtype B

\Rightarrow B substituted by A

e.g.: I32 < I64

\Rightarrow B from A in call OK

i.e. can pass I32 for I64 w/o losing info

e.g.: $\{x, y, u, \dots T\} < \{x, y, T\}$

$\Leftrightarrow T < u$

i.e. can pass $\{x, y, u, \dots T\}$

as $\{x, y, T\}$

$\Leftrightarrow T < u$

e.g. I32 < Float64

i.e. can pass I32 as float64

* A sub B

\Leftrightarrow if $f(B) \rightarrow x$, then

$f(A) \rightarrow x'$ [s.t. x' sub x]

* Rule :

① Int \rightarrow Float :

$I8 \leq I16 \leq I32 \leq I64$

$U8 \leq U16 \leq U32 \leq U64$

$F32 \leq F64$

② Struct :

$\{ a_1 \dots a_m \mid T_1 \dots T_m \}$

\vdots

$\{ b_1 \dots b_n \mid U_1 \dots U_n \}$

\Downarrow

① $m \geq n$ (sub bigger)

② $\forall 1 \leq i \leq m,$

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

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

$T_i \leq U_j$

all names unique

$\{ x \mid I32 \}$

\vdots

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

③ Enum

Each enum accepts a struct!

Follow struct typing rules.

④ VFunc

$A \dots \rightarrow B \leq C \dots \rightarrow D$

\Updownarrow

(A)

$\text{Im}(A) = \text{Im}(C) \quad \textcircled{1} \quad \forall i \quad A_i \leq C_i$

⑤ TFunc

see VFunc, Class

* ①

Subclass

$C \leq D$

$\Leftrightarrow \forall \tau \in C, \tau \in D$

$\Leftrightarrow C \subseteq D$

for calls,

not really

* consider sep. subclasses / 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:

 m F32;

}

class Y {

public:

 extend base X;

 m F32;

}

 ~ more! public field "m" overridden