
ORACLE

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

ORACLE®

Dynamic overloaded dispatch and generic type inference in Fortress

David Chase (presenter), Justin Hilburn, Victor Luchangco, Karl Naden, Guy Steele, Jean-Baptiste Tristan

Programming Languages Research Group, Oracle Labs

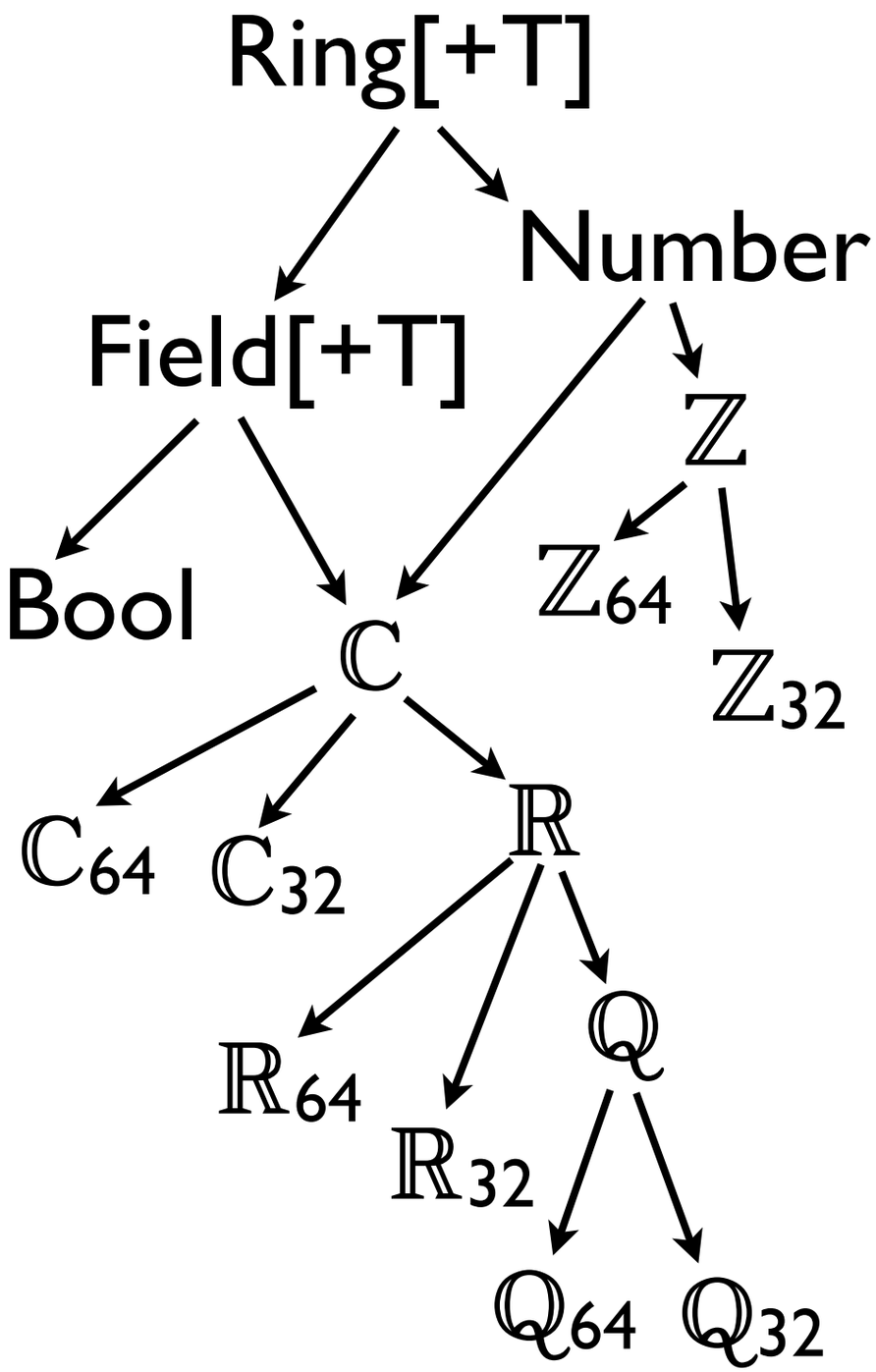
Fortress

- "Run your white board, in parallel!"
- Space-sensitive mathematical syntax
- Work-stealing load balancing
- Parallel nested transactions
- Generic functions, methods, traits, objects
- Multi-arg dispatch overloaded functions
- Statically checked, dynamically inferred

Java vs. Fortress dispatch

```
class Foo {
   String data;
   public boolean equals(Foo x) {
      return data.equalsIgnoreCase(x.data);
   }
   // Fortress dispatch
   public boolean equals(Object x) {
      return false;
   }
   // Java dispatch
   public boolean equals(Object x) {
     return (x instanceof Foo) ?
             equals((Foo) x) :  false;
   }
}
```

Generic overloading uses



TIMES[T <: Ring[T]]
(x: Mat[T], y:Mat[T]):Mat[T]

(x: Diag[T], y:Mat[T]):Mat[T]

(x: Mat[T], y:Diag[T]):Mat[T]

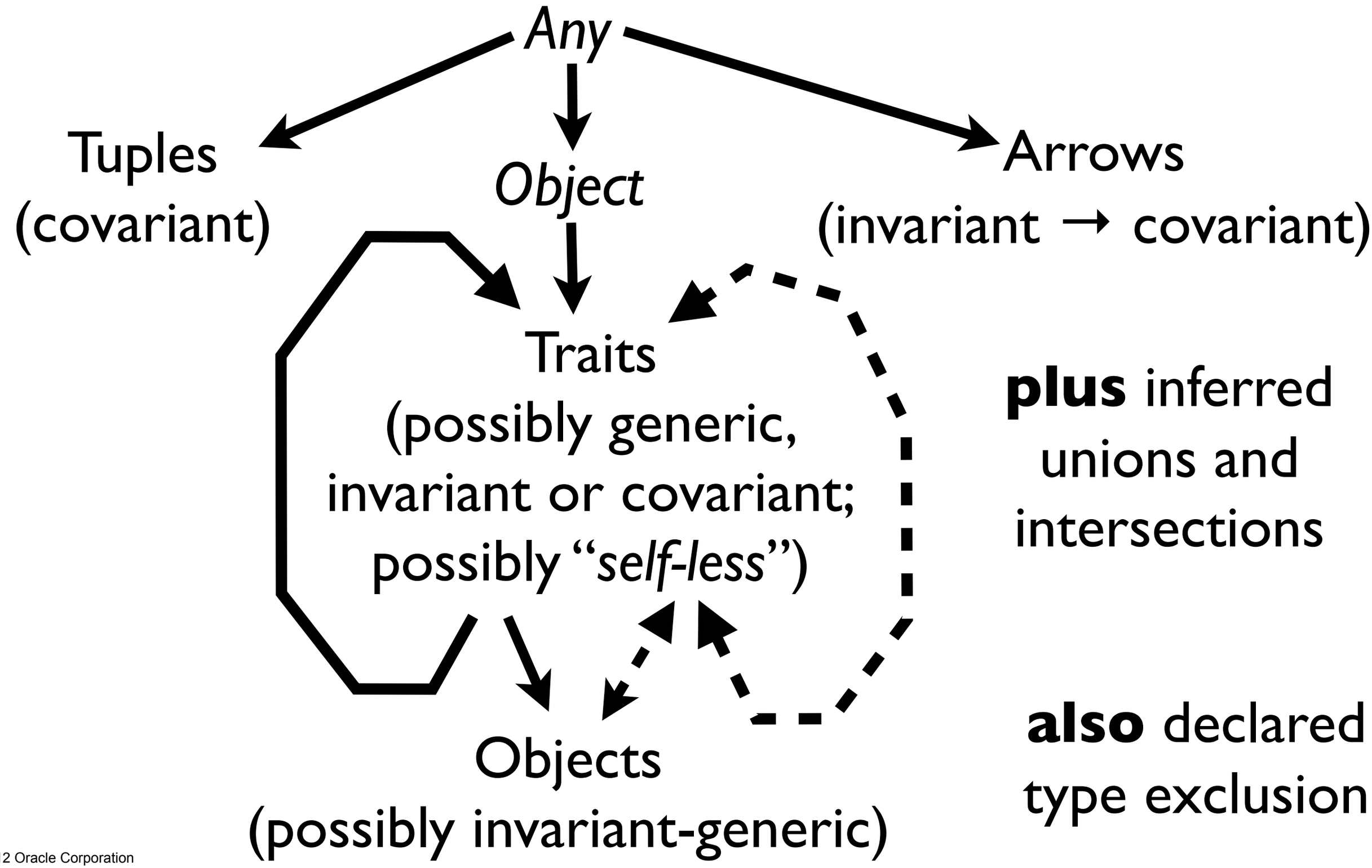
(x: Diag[T], y:Diag[T]):Diag[T]

(x: Mat[Bool], y:Mat[T]):Mat[T]

(x: Diag[Bool], y:Mat[T]):Mat[T]

(x: Diag[Bool], y:Diag[T]):Diag[T]

Quick type system



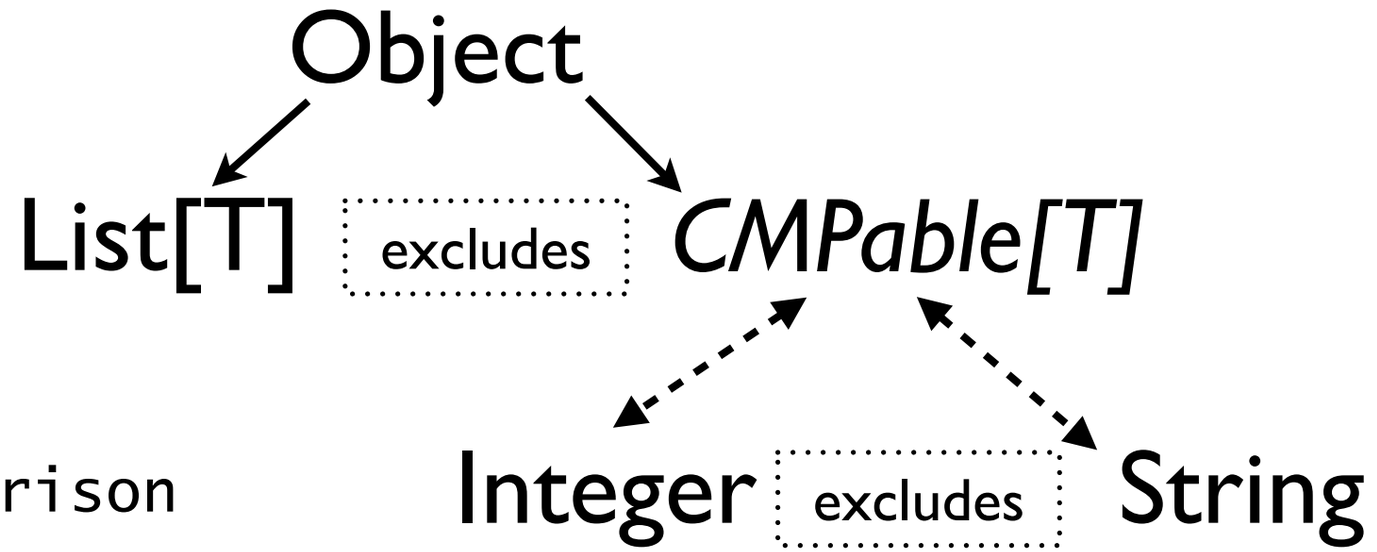
plus inferred unions and intersections

also declared type exclusion

Self-less generic types

```
trait CMPable[T] comprises T
   abstract opr < (self, other:T):Boolean
   opr > (self, other:T):Boolean = other < self
   opr = (self, other:T):Boolean =
                        ¬ ( self < other ∧ other < self )
   opr CMP (self, other:T):Comparison =
      if self < other then LessThan
      elif self > other then GreaterThan
      else Equals
      endif
end
```

Example




```
CMP(x:String, y:String):Comparison
CMP(x:Integer, y:Integer):Comparison
CMP[T extends CMPable[T]](x:CMPable[T], y:CMPable[T]): Comparison
CMP[T extends CMPable[T]](x:List[T], y:List[T]): Comparison
CMP(x:Object, y:Object):Comparison
```

CMP is overloaded, and includes generic entrypoints, and the outer type (Obj, Obj)→Comparison is not generic.

Sort entrypoints into a most-to-least specific order and sequentially *test applicability*. First success is best choice.

“Test applicability?”

- Pattern-match the argument type with the entrypoint’s signature type.
- Accumulate initial constraints on parameterized type occurrences in the signature.
- Propagate constraints across type parameters.
- Succeed if constrained sets are nonempty.

Is this well-defined? Is it efficient?

Type system restrictions

- All paths to Any finite
- Unions and Intersections inferred, **not** declared
- For each generic supertype, a minimum instantiation must be declared
- Each “self-less” trait is singly-inherited
- Function type constraints are scoped/ordered, with form declared-parameter-extends-expr
- NO CONTRAVARIANCE
  - But we have a workaround for “useful” cases

Pattern-match and accumulate constraints

actual (ground) type of function arguments

entrypoint signature (includes type parameters)

compile-time constants for pattern-matching

```
1:  function MATCH(g:Type, V:Variance, s:Type)
2:      if s is a ground type then
3:          if V ≥ 0 then    ⊛ co/invariant, require g <: s
4:              if ¬(g <: s) then  dispatch fails
5:          if V ≤ 0 then    ⊛ contra/invariant, require s <: g
6:              if ¬(s <: g) then  dispatch fails
7:      else if s is a type parameter P then
8:          if V ≥ 0 then    ⊛ co/invariant, require g <: P
9:              insert g into L_P
10:         if V ≤ 0 then    ⊛ contra/invariant, require P <: g
11:             insert g into U_P
12:     else if s is an arrow type s_domain → s_range then
13:         if g is an arrow type g_domain → g_range then
14:             MATCH(g_domain, 0, s_domain)
15:             MATCH(g_range, V, s_range)
16:         else dispatch fails
17:     else if s is a tuple type (s_1, ... , s_m) then
18:         if g is a same-length tuple type (g_1, ... , g_m) then
19:             for 1 ≤ j ≤ m do
20:                 MATCH(g_j, V, s_j)
21:         else dispatch fails
22:     else if g is a union of types g_1, ... , g_m then
23:         ⊛ s is a constructed signature type
24:         if V = 0 then  dispatch fails
25:         else
26:             for 1 ≤ j ≤ m do
27:                 MATCH(g_j, V, s)
28:     else
29:         ⊛ s is a constructed signature type T⟦s'_1, ... , s'_m⟧
30:         ⊛ g is not a union or intersection
31:         if g has some ancestor of the form T⟦_⟧ then
32:             let g' = the minimal ancestor T⟦g_1, ... , g_m⟧ of g
33:                 that has the form T⟦_⟧
34:             if (V = 0) ∧ (g ≠ g') then  dispatch fails
35:             let (V_1, ... , V_m) = variances of T's parameters
36:             for 1 ≤ j ≤ m do
37:                 MATCH(g_j, V × V_j, s'_j)
38:         else dispatch fails
```

an interesting part

29: ⊛ s is a constructed signature type $T[\![s'_1, \ldots, s'_m]\!]$
30: ⊛ g is not a union or intersection
31: **if** g has some ancestor of the form $T[\![\_]\!]$ **then**
32: **let** g' = the minimal ancestor $T[\![g_1, \ldots, g_m]\!]$ of g
33: that has the form $T[\![\_]\!]$
34: **if** $(V = 0) \wedge (g \neq g')$ **then** dispatch fails
35: **let** $(V_1, \ldots, V_m)$ = variances of T's parameters
36: **for** $1 \leq j \leq m$ **do**
37: MATCH$(g_j, V \times V_j, s'_j)$
38: **else** dispatch fails

Propagate constraints

Least Single Upper Bound

Trivial if just one

1: **for** j sequentially from n down to 1 **do**
2: $l_j \leftarrow \text{LSUB}(\mathcal{L}_{P_j})$
3: iterate up from l_j through self-less constraints
4: to simultaneous solution
5: **if** $l_j \not<:$ Bottom **then**
6: **for** each declared self-ish upper bound ξ of P_j **do**
7: **if** ξ is a parametric signature type **then**
8: $\text{MATCH}(l_j, +1, \xi)$
9: **for** each u in $\mathcal{U}_{P_j}$ **do**
10: **if** $l_j \not<: u$ **then**
11: dispatch fails

|U_Pj| = 0 case can be spotted at compile time.

Worked example

Suppose actual inputs to CMP are a pair of List[String], e.g.,
x = < “cat”, “dog” > and y = < “bat”, “rat” >

CMP(x:String, y:String)

```
IF NOT( g subtype-of (String, String) )
THEN dispatch fails
ENDIF
```

CMP(x:Integer, y:Integer): Comparison

```
IF NOT( g subtype-of (Integer, Integer) )
THEN dispatch fails
ENDIF
```

CMP[T extends CMPable[T]] (x:CMPable[T], y:CMPable[T]): Comparison

```
IF g is a tuple type (g_1, g_2) THEN
// Recursive calls expanded
  IF g_1 is a union THEN ...
  ELSIF g_1 extends CMPable[something] THEN ...
  ELSE dispatch fails
  ENDIF
  IF g_2 is a union THEN ...
  ELSIF g_2 extends CMPable[something] THEN ...
  ELSE dispatch fails
  ENDIF
ELSE dispatch fails
ENDIF
...
```

Example cont.

```
CMP[T extends CMPable[T]] (x:List[T], y:List[T]): Comparison
      IF g is a tuple type (g_1, g_2) THEN
        IF g_1 is a union THEN ...
        ELSIF g_1 extends List[g_1_a] THEN
          // match (g_1_a, 0, T) // invariant List
          insert g_1_a into L_T
          insert g_1_a into U_T
        ELSE dispatch fails
        ENDIF
        IF g_2 is a union THEN ...
        ELSIF g_2 extends List[g_2_a] THEN
          // match (g_2_a, 0, T) // invariant List
          insert g_2_a into L_T
          insert g_2_a into U_T
      ELSE dispatch fails
      ENDIF
      ENDIF
      l_T := LSUB(L_T)
      // Self-less constraint check
      IF l_T extends CMPable[l_T'] THEN l_T := l_T'
      ELSE dispatch fails
      ENDIF
      // No self-ish constraints
      FOR u IN U_T DO
         IF NOT l_T extends u THEN dispatch fails ENDIF
```

```
g_1 = List[String],
g_2 = List[String]

g_1_a = String

L_T = {String}
U_T = {String}

g_2_a = String

L_T = {String}
U_T = {String}

l_T := String

l_T := String
```

Different input

x = < “cat”, “dog” > and y = < 3, 4 >

CMP[T extends CMPable[T]] (x:List[T], y:List[T]): Comparison

```
      IF g is a tuple type (g_1, g_2) THEN
        IF g_1 is a union THEN ...
        ELSIF g_1 extends List[g_1_a] THEN
          // match (g_1_a, 0, T) // invariant List example
          insert g_1_a into L_T
          insert g_1_a into U_T
        ELSE dispatch fails
        ENDIF
        IF g_2 is a union THEN ...
        ELSIF g_2 extends List[g_2_a] THEN
          // match (g_2_a, 0, T) // invariant List example
          insert g_2_a into L_T
          insert g_2_a into U_T
        ELSE dispatch fails
        ENDIF
      ELSE dispatch fails
      ENDIF
      l_T := LSUB(L_T)
      // Self-less constraint check
      IF l_T extends CMPable[l_T’] THEN l_T := l_T’
      ELSE dispatch fails
      ENDIF
      // No self-ish constraints
      FOR u IN U_T DO
         IF NOT l_T extends u THEN dispatch fails ENDIF
```

```
g_1 = List[String],
g_2 = List[Integer]

  g_1_a = String

  L_T = {String}
  U_T = {String}

  g_2_a = Integer

L_T = {String, Integer}
U_T = {String, Integer}

l_T := String U Integer

  FAIL!
```

Run-time type operations

- A subtype B? (what if A, B generic?)
- A join/lsub B
- A instanceof G[T]?, minimum value of T

Longest Erased Path to Any

- LEPA(Any) = 0,
  LEPA(Object) = 1,
  LEPA(trait T) =
  1 + max({LEPA(S) | T extends S})
- Syntactic definition, straight from declarations.

Supers(T)

LEPA	Supers($\mathbb{R}$)	Supers($\mathbb{C}$)
0	Any	Any
1	Object	Object
2	Ring→[$\mathbb{R}$] Number	Ring→[$\mathbb{C}$] Number
3	Field→[$\mathbb{R}$]	Field→[$\mathbb{C}$]
4	$\mathbb{C}$	$\mathbb{C}$
5	$\mathbb{R}$	

A subtype B = LEPA(A) >= LEPA(B) $\wedge$

Stem(B) $\in$ Supers(A)[LEPA(B)] $\wedge$

{ compare Supers(A)[LEPA(B)].args with B.args }

JVM issues

- Current implementation makes heavy use of applicative caches (declared volatile, CAS for update, no-lock reads). Volatile inhibits optimization by JIT.
- Type data structures are initialized lazily (in a concurrent context), but then never change.
- Should we use method handles and invokedynamic?
- Type unions, intersections, tuples, and arrows require special treatment; cannot practically be pre-declared as supertypes.
- Our "type constants" might not look like constants to the JIT optimizer.
- Some (important) optimization opportunies for dispatch itself appear after only generic instantiation (AT RUNTIME)

JVM changes?

- Lighter-weight volatile? (No. JMM scary!)
- Write-once variables (lazy final)
- Idempotent (pure-mostly) methods? (Known tech)
- Interface injection? (Make more fast paths, allow better mapping onto JVM types)
- Unerased generics? (Supporting "our" type operations? or not?)

Questions?

Likely optimizations

- Indexed (vtable) dispatch as first step into various alternatives.
- Splitting dispatch search when partial order contains articulation points.
- Redundancy elimination across dispatch tests.
- Clause reordering (common first)
- Type erasure or non-canonicalization as an optimization (type bindings not always used).

Contravariance-lack-hack

Replace each nominally contravariant occurrence of T with T_k, adding invariant T_k before T in the static parameter list, and the constraint "T extends T_k".

Example:

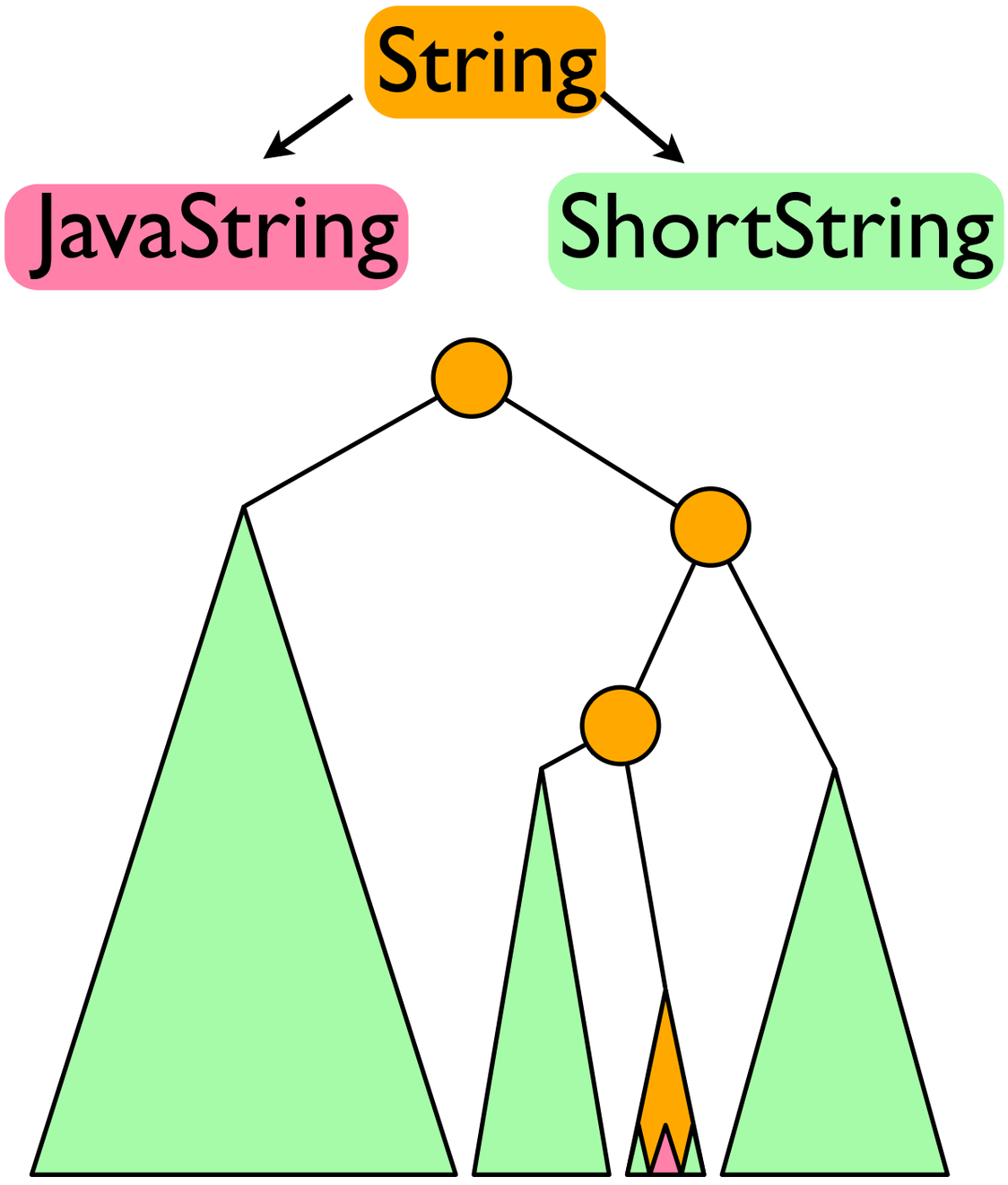
map[T, U](l:MutList[T], f:T->U) : List[U]

rewrites to

map[T_a, T<:T_a, U](l: MutList[T], f:T_a->U) : List[U]

Dynamic inference uses

ApplBalBinTree [covariant T extends CMPable[T]]



An object (leaf) type.
23 or fewer 8-bit non-zero chars.
Immediate data, fast compares.

```
lookup[ K extends CMPable[K], V ]
    (key:K, tree:ABBT[ K,V ] ): Maybe[V] = do
 c = key CMP tree.key
 if c = Eq then Some(tree.value)
 elif c = Lt then if tree.left.isSome
   then lookup(key, tree.left.unwrap)
   else None[V] end
 elif tree.right.isSome
   then lookup(key, tree.right.unwrap)
   else None[V] end
end end
```

Handling intersections

1: $\langle$*intersection subtype of constructed signature type?*$\rangle$ =
2: **else if** g is an intersection of types $g_1, \ldots, g_m$ **then**
3: ⊛ s is a constructed signature type
4: ⊛ n is the number of method type parameters
5: **if** $V = 0$ **then** dispatch fails
6: **else**
7: **for** $1 \leq i \leq n$ **do**
8: Save existing bound sets $\mathcal{L}_{P_i}$ and $\mathcal{U}_{P_i}$
9: $\mathrm{LBT}_{P_i} \leftarrow \mathrm{Any}$ ⊛ Lower bound type
10: $\mathrm{UBT}_{P_i} \leftarrow \mathrm{Bottom}$ ⊛ Upper bound type
11: **var** *anymatch* = *false*
12: **for** $1 \leq j \leq m$ **do**
13: **for** $1 \leq i \leq n$ **do**
14: $\mathcal{L}_{P_i} \leftarrow \{\}$
15: $\mathcal{U}_{P_i} \leftarrow \{\}$
16: **try**
17: MATCH(g_j, V, s) ⊛ sole use of j is here
18: **for** $1 \leq i \leq n$ **do**
19: $\mathrm{LBT}_{P_i} \leftarrow \mathrm{LBT}_{P_i} \cap \bigcup_{g' \in \mathcal{L}_{P_i}} g'$
20: $\mathrm{UBT}_{P_i} \leftarrow \mathrm{UBT}_{P_i} \cup \bigcap_{g' \in \mathcal{U}_{P_i}} g'$
21: *anymatch* $\leftarrow$ *true*
22: **catch** dispatch failure
23: **if** $\neg$*anymatch* **then** dispatch fails
24: **for** $1 \leq i \leq n$ **do**
25: Restore saved bound sets $\mathcal{L}_{P_i}$ and $\mathcal{U}_{P_i}$
26: **if** $\mathrm{Any} \not<: \mathrm{LBT}_{P_i}$ **then**
27: $\mathcal{L}_{P_i} \leftarrow \mathcal{L}_{P_i} \cup \{\mathrm{LBT}_{P_i}\}$
28: **if** $\mathrm{UBT}_{P_i} \not<: \mathrm{Bottom}$ **then**
29: $\mathcal{U}_{P_i} \leftarrow \mathcal{U}_{P_i} \cup \{\mathrm{UBT}_{P_i}\}$

$\bigcap$ g_i <: A[T], true if one or more g_i <: A[T]. Largest set of g_i gets most specific result.

Relies on distribution of intersection into generics.

Proof that intersection distributes

Suppose G[covariant P].

A ∩ B <: A, A ∩ B <: B,

therefore G[A ∩ B] <: G[A], G[A ∩ B] <: G[B],

therefore G[A ∩ B] <: G[A] ∩ G[B].

Suppose G[A] ∩ G[B] </: G[A ∩ B].

Then there is a v in G[A] ∩ G[B] not in G[A ∩ B], and v's ilk is an object type V. V <: G[A], V <: G[B],

therefore there is T such that T <: A, T <: B, V<: G[T].

But T <: A ∩ B, implying that v in V <: G[T] <: G[A ∩ B], a contradiction. And T is not bottom, because G[T] was declared as a supertype of V.

Self-less example

```
trait Foo[\ T \] comprises T
  opr +(self, y: T): T
  opr -(self): T
  opr -(self, y: T): T = self + (-y)
  double(self): T = self + self
end
```

```
object Bar(content: ZZ32) extends Foo[\ Bar \]
  opr +(self, y: Bar): Bar = Bar(content + y.content)
  opr -(self): Bar = Bar(-content)
end
```