Type System

Type := NominalType | Array | TArray | Dynamic | ProceduralType | IntersectionType | SymbolType
 | UnionType | FieldType
NominalType := Fixnum |String | TrueClass | FalseClass | Object | …
ProceduralType := [(arg$_1$, … arg$_n$) -> (NominalType U Array U TArray U Dynamic)]
 | [IntersectionType(ProceduralType$_1$,… ProceduralType$_n$)]
arg$_i$:= Type\IntersectionType
NominalType – the simplest kind of type, where the name is the same as the type, such as Fixnum, String, Object, etc.

Polymorphic Checking with Union Types

$$\frac{t = typeA \in arg_i \qquad t = typeB \in arg_j}{t = typeA \cup typeB}$$

f<t>: (t, t) -> Fixnum
f(1, "str") = 0
You get constraints:

t >= Fixnum
t >= Fixnum U String

But this is bad because you don't get an error even though the two types are different.
If the programmer wants to enforce the two t's are the different, then he could change one of the t's to an u.

Polymorphic Checking with Open Types

Constraint Solving

We generate a constraint for each method argument where the typesig for that argument contains a parameter. Similarly, we do this for return types.

We take the constraints and get a map where the each key is a parameter symbol and the corresponding value is the corresponding type. First, we solve the trivial cases. These are the cases where there are no Unions.

For example, suppose we have constraints:

t or u >= Fixnum or String
t >= Fixnum

Then we first get the solution t = Fixnum, and eliminate the second constraints. We also modify the first constraint to "Fixnum or u >= Fixnum or String". Here we can conclude u = String.

However, suppose we have subtyping relations C <= B <= A and the following constraints:

t >= C
t or u >= B
t or u >= A
u >= X

The correct solution here is t = A and u = X. So, we cannot just take "t = C" as the solution for t. These are some cases to consider

1) First, we do store "t = C" as the solution for t, as well as "u = X". But, when we get to the constraint "t or u >= B" → "C or X >= B", we consider "t = C" invalid and update t to B. Similarly, we get perform this t update for "t or u >= A". If there are other constraints including unions of t, then we may have many backtracks.
2) First, we know that "t >= C", but t is also used in unions in other constraints. So we do not initially consider t in the solution. We see that "u >= X" and even though u is used in other unions, the RHS of both unions belong to a classes that are unrelated to X. So we take "u = X" as the first solution. Then, we may still have to do backtracking depending on which constraint we get to next.

Now, in a case like
f<t, u>: (Array<t> or u, Array<t> or u, …) → …
f ([1,2], [3,4], …)

Should we say t = Fixnum ^ u = Fixnum OR
t = Array<Fixnum> ^ u = Array <Fixnum> OR
t = Fixnum ^ u = Array <Fixnum> OR …?

Finding the correct solution may involve trying every possible combination.

Exactly what does the programmer expect?

<u>Various Examples with Open Types</u>

1) my_push<t>: (Array<t>, t) -> t
   my_push([1,2], "str") -> [1,2,"str"]

Constraints (without open types):
t >= Array<Fixnum>, t >= String, t >= Array<Fixnum or String>

Without the use of open types, the following will fail:
my_push([1,2,3], “foo”)
But this will succeed:
my_push([1,2,3,”foo”], “bar”)

The former is correct if t has not been constrained to Array<t>

Another simple solution would be to make the programmer change the typesig to
my_push<t, u>: (Array<t>, u) -> Array<t or u>
But this is not a very flexible typesig.

With open types, we get t >= [* Fixnum], t >= String, t >= [* Fixnum U Strings]
[* …] indicates the open type, and the first two constraints can be combined to t >= [* Fixnum].

2) my_method<t>: (Array<t>, Array<t>) -> whatever

Should this fail?
my_method([1,2].rtc_annotate(“Array<Fixnum>”), [1, “foo”])

Naively we say t = Fixnum U String. But then we have some u and assign Array<t> = u.
Now things get weird because the first argument is not a subtype of u, but the second is.