I am now looking into algorithm typing of DOT and following are my thoughts from last week. Let me know if you have suggestions. To summarize, my questions are following, and details are broken down into further below.

1. Is it a good idea to unify record type and object type?
2. To handle subtyping, it does not seem straightforward to me to get around computing SCC. Is it good enough to assert "facts" about SCC, since it's well known?
3. Up to which point is it sufficient in terms of handling and updating SCCs?

Detailed discussions are following:

1. treatment of recursive type.

   In order to turn type judgment into an algorithm, we need to eliminate non-determinism to a point where there is only one rule for each context-term pair. Essentially, function with following signature is supposed.

   ```
   typeJudge :: Context -> Term -> Type
   ```

   Potentially, this function should return the most precise type that can get from the context.

   I find that following two judgments are especially problematic, since they can be applied to all terms.

   $$\frac{\Gamma \vdash x : T}{\Gamma \vdash x : \mu(x : T)}\ \textbf{(REC-I)} \qquad \frac{\Gamma \vdash x : \mu(z : T)}{\Gamma \vdash x : [x/z]T}\ \textbf{(REC-E)}$$

   I think one way to deal with these two is to remove them. My rationale is, the existence of two rules simply indicates the essential equivalence of any type and its recursive wrapper. But if I do this, there will be difficulties to handle object when a record is required, and the other way, even though any record type can only be instantiated by objects, and in reality, there really isn't fundamental difference between them.

   Following this line of thought, it seems that unifying object type and record type is an option. However, that really means the model language will be changed. Do you think it's something OK to do? It sounds to me doing so becomes finding an alternative model language already. But if not doing so, it's not obvious to me that if there exists other algorithmic way to handle these two rules.

   One observation of looking to unifying these two types is, the type of resulting language will essential become top + bottom + structural refinement + lambdas + paths, which shows some similarities between both DOT and featherweight Scala. And we've known the latter has algorithmic typing.

2. subtyping.

   I feel some difficulties regarding this part as well. Following the similar path to step typing, I am thinking in terms of three functions for this part.

   ```
   isSubType :: Context -> Type -> Type -> Bool
   exposure :: Context -> Type -> Type
   promote :: Context -> Var -> Type -> Type
   ```

   The motivation of thinking of these two functions, following the same path of step typing, is to reformulate following rules.

   $$\frac{\Gamma \vdash x : T' \qquad (\mathit{exposure}\ \Gamma\ T') = \forall(z : S)T \qquad \Gamma \vdash y : S' \qquad \mathit{isSubType}\ \Gamma\ S'\ S}{\Gamma \vdash x\ y : [y/z]T}\ \textbf{(ALL-E)}$$

   $$\frac{\Gamma \vdash t : T \qquad \Gamma, x : T \vdash u : U}{\Gamma \vdash \mathit{let}\ x = t\ \mathit{in}\ u : (\mathit{promote}\ (\Gamma, x : T)\ x\ U)}\ \textbf{(LET)}$$

   However, regardless of whether we change the model language or not, there is a cycle issue among subtypes. So I think SCC at minimum is necessary. But some search shows that even implementing DFS would not be trivial. http://gallium.inria.fr/~fpottier/publis/fpottier-dfs-scc.pdf this paper shows DFS as a function, but not yet reach SCC. Even though one exists, properties involving SCC might not be straightforward. Are you aware how people do formalization with cycle detection? Is it OK to assert axioms to get away with proving those well-known properties of SCC?

3. maintaining SCC.

   Thinking ahead, I am trying to obtain a big picture of how the graph representing subtype relation should be maintained. I am think of following two (dual) functions in order to define **exposure** and **promote**:

```
    -- |the supertype of all types that are subtypes of two input types.
    supremum :: Context -> Type -> Type -> Type
    -- |the subtype of all types that are supertypes of two input types.
    infimum :: Context -> Type -> Type -> Type
```

   To do this, I imagine whenever an object type is pushed into the typing context, the graph needs to be updated, and the function **exposure** and **promote** will rely on the graph. I am thinking of the update in terms of following steps(an edge $U \to V$ indicates $U <: V$ in the context).

Algorithm 1 graph update algorithm

```
1: for $\{A : S..T\}$ in $\mu(x : \{...\})$ do
2:     add $S \to x.A$, $x.A \to T$ to the graph
3: end for
4: run SCC on the new graph
5: /*following step guarantees every path type has a concrete type as its infimum and supremum for every SCC.   */
6: for (weakly) connected SCC subgraph in SCC graphs do
7:     add edges from ⊤ to the sources that are a path type
8:     add edges to ⊥ from the sinks that are a path type
9: end for
```

   The second for loop is to handle those object types which purely introduce cycles.

```
    μ x : {
      A : x.B .. x.C
      B : x.C .. x.A
      C : x.A .. x.B
    }
```

$$\begin{tikzcd}
 & \top & \\
 & A \arrow[u] \arrow[rd, bend left] & \\
B \arrow[ru, bend left] & & C \arrow[ll, bend left] \\
 & \bot \arrow[u] &
\end{tikzcd}$$

   Here, the cycle collapse all these three types into a single one, and they can be assigned to any types. I can think of two ways to handle each SCC.

   (a) Leave the SCC as it is. This in fact simulates what DOT actually does. However, since DOT is non-deterministic, it's unclear to me how to make **promote** or **exposure** deterministic and return the most "desirable" type. Also the cycle might contain inconsistency, like following.

```
    μ x : {
      A : ∀(y: ⊥)⊤ .. μ z: { C : ⊥ .. ⊤ }
      B : μ z: { C : ⊥ .. ⊤ } .. ∀(y: ⊥)⊤
    }
```

   (b) One observation is, assuming no intersection type is involved, each SCC degenerates to simply typed lambda calculus with objects, and therefore unification seems possible with all path types treated as type variables. Therefore all non path types need to be syntactically compatible. This effectively tries hard to prove the type lattice remains familiar, after the subtype constraints are added.

   There are subtleties in this method as well. Since every unification will introduce new equations, after this, we need to keep doing unification with new equations until the number of SCCs stablizes, the termination of which is not obvious at all. It's more complex than simply require $S <: U$ being provable for each $\{A : S <: U\}$ like it's done in $D_{<:}$, due to the recursive nature of object type.

It seems to me that this constraint resolution part will be big and with all these heavy dependencies on graph theory, I am not sure if it can be formalized in coq. Do you have better suggestions on how to approach it?

Will it be beneficial to understand how dotty handles paths? For now, I am trying to look into order theory to get some inspiration.

4. intersection type.

   I think intersection type is very difficult to handle, as it introduces too much non-determinism if intersection types involve path types. For example, it prevents unification like above from being possible.

```
  μ x : {
    A : ⊥ .. ⊤
    B : ⊥ .. ⊤

    C : x.A ∧ x.B .. μ x : { ... }
    D : μ x : { ... } .. x.A ∧ x.B -- same object type as above
  }
```

   It's entirely unclear what information can be retrieved from $x.A \wedge x.B \equiv \mu\ x : \{\ ...\ \}$. In this case, $x.A$ can be anywhere from $\top$ to all of $\mu\ x : \{\ ...\ \}$. More generally, for relation $A \wedge B <: C$, the only derivable relation from this is $A \wedge B <: A \wedge C$ and $A \wedge B <: B \wedge C$, which gives no better information than orginally. I think the interaction between intersection type and path type might become the source the undecidability, if one ever proves it. Considering all the previous problems are quite complex already, I am thinking leave this case. What do you think about this?