We have learned how we can incorporate the substitution principle of object-oriented languages into our type system by extending it with a subtype relation. It remains to discuss how we can actually implement a type inference algorithm for the new typing relation. The problem with the current definition of the subtype relation and the subtyping rule is that these rules are not syntax-directed. Hence, when we try to implement the type inference algorithm by reading the inference rules bottom-up, we cannot decide how to apply the new rules just by looking at the syntactic structure of expressions. For example, reconsider the subtyping rule itself:

\[ \Gamma \vdash e : \tau' \quad \tau' \ll : \tau \quad \text{TypeSub} \]

Note that the rule applies to any expression \( e \) without restrictions on \( e \)'s syntactic structure. For each top-level syntactic constructor used in a program expression, our typing relation also has an ordinary rule that does not involve subtyping. Given an expression \( e \), we could use this ordinary typing rule to infer the type \( \tau' \) of \( e \) that occurs in the premise of the rule TypeSub in an attempt to compose the two rules together. However, even if we did that, we would still need to guess the supertype \( \tau \) of \( \tau' \) in the rule TypeSub. How can we implement this rule without having to guess any of the types involved?

For the subtype relation itself the situation is similar. Consider the rule for transitivity of subtyping:

\[ \tau_1 \ll : \tau_2 \quad \tau_2 \ll : \tau_3 \quad \text{SubTrans} \]

In order to apply this rule during type inference (reading the rule bottom-up), we would have to guess the intermediate type \( \tau_2 \) to make the transitive step between \( \tau_1 \) and \( \tau_3 \). Again, it is unclear how to implement this rule without having to guess any types.

We will solve these problems by modifying the subtyping and typing rules so that they become again syntax-directed.
\[ \tau <: \tau \quad \text{SubRefl} \]
\[
\frac{\tau_2 <: \tau'_2 \quad \tau'_1 <: \tau_1}{(x:\tau_1) \Rightarrow (y:\tau'_1) \Rightarrow \tau_2} \quad \text{SubFun}
\]

\[ \{g_1, \ldots, g_m\} \subseteq \{f_1, \ldots, f_n\} \]

For all \( i, j \), if \( f_i = g_j \), then \( \text{mut}_i = \text{mut}'_j = \text{const} \) and \( \tau_i <: \tau'_j \)

or \( \text{mut}_i = \text{mut}'_j = \text{var} \) and \( \tau_i = \tau'_j \)

\[ \{\text{mut}_1 f_1:\tau_1; \ldots; \text{mut}_n f_n:\tau_n\} <: \{\text{mut}'_1 g_1:\tau'_1; \ldots; \text{mut}'_m g_m:\tau'_m\} \quad \text{SubObj} \]

Figure 1: Algorithmic subtyping rules for JakartaScript

Syntax-Directed Subtyping

We start by redefining the subtype relation. Let us first analyze how the transitivity rule is used in a subtyping derivation. To this end, reconsider the example derivation from previous class:

\[ \{\text{const} g: \text{bool}; \text{var} f: \text{bool}\} <: \{\text{var} f: \text{bool}; \text{const} g: \text{bool}\} \quad \text{SubObjPerm} \]

\[ \{\text{var} f: \text{bool}; \text{var} g: \text{bool}\} <: \{\text{var} f: \text{bool}\} \quad \text{SubObjWidth} \]

\[ \{\text{const} g: \text{bool}; \text{var} f: \text{bool}\} <: \{\text{var} f: \text{bool}\} \quad \text{SubTrans} \]

We can see that we use the transitivity rule to combine several applications of the object subtyping rules—here, the rules SubObjPerm and SubObjWidth. It turns out that these are the only situations where the transitivity rule is needed. Thus, we can eliminate the transitivity rule by merging it with the object subtyping rules SubObjPerm, SubObjWidth, and SubObjDepth into a single rule for object subtyping. The new subtyping rule is shown in Figure 1. The new rule SubObj combines all the previous object subtyping rules with the transitivity rule.

Now, all of our subtyping rules are syntax-directed. By looking at the structure of the typing expression, we know which rule to apply. Note that the rule SubRefl can be restricted to the cases where \( \tau \) is one of the basic types number and bool.

Syntax-Directed Typing with Subtyping

Our solution for making our typing relation syntax-directed is similar: we eliminate the subtyping rule Typesub by merging it with those syntax-directed typing rules where subtyping is really needed. The modified typing rules are shown in Figure 2. For the syntactic primitives where no new rule is provided, the old typing rule still applies. We discuss the new typing rules in more detail.

Function Calls. The rule TypeCall is the new rule for typing function call expressions. Observe that the new rule no longer requires that the argument
\[
\Gamma \vdash e_1 : (x : \tau_2) \Rightarrow \tau \\
\quad \Gamma \vdash e_2 : \tau_2' \\
\quad \tau_2' \ll \tau_2 \\
\quad \Gamma \vdash e_1(e_2) : \tau
\]

\text{TypeCall}

\[
\Gamma' = \Gamma[x \mapsto (\text{const}, \tau_2)] \\
\Gamma' \vdash e : \tau' \\
\quad \tau' \ll \tau
\]

\text{TypeFunAnn}

\[
\begin{array}{l}
\Gamma' = \Gamma[x_1 \mapsto \tau_1][x_2 \mapsto (\text{const}, \tau_2)] \\
\tau_1 = (x_2 : \tau_2) \Rightarrow \tau \\
\Gamma' \vdash e : \tau' \\
\quad \tau' \ll \tau
\end{array}
\]

\text{TypeFunRec}

\[
\begin{array}{l}
\Gamma(x) = (\text{var}, \tau') \\
\Gamma \vdash e : \tau
\quad \tau \ll \tau'
\end{array}
\]

\text{TypeAssignVar}

\[
\begin{array}{l}
\Gamma \vdash e_1 : \{ \ldots \text{var } f : \tau' \ldots \} \\
\Gamma \vdash e_2 : \tau \\
\quad \tau \ll \tau'
\end{array}
\]

\text{TypeAssignFld}

\[
\begin{array}{l}
\Gamma \vdash e_1 : \text{bool} \\
\Gamma \vdash e_2 : \tau_2' \\
\Gamma \vdash e_3 : \tau_3 \\
\quad \tau_2' \sqcup \tau_3 = \tau
\end{array}
\]

\text{TypeIf}

\[
\begin{array}{l}
\Gamma \vdash e_1 : \tau_1 \\
\Gamma \vdash e_2 : \tau_2 \\
\quad \tau_1 \sqcup \tau_2 = \tau
\end{array}
\]

\text{TypeEqual}

\[
\begin{array}{l}
\tau_1 \text{ has no function types} \\
bop \in \{==, !=\}
\end{array}
\]

\text{TypeEqual}

Figure 2: Typing rules with subtyping
type and parameter type of the called function are equal. Instead, the rule now incorporates the subtyping rule directly by requiring that the argument type can be a subtype of the parameter type.

Type Annotations. In the cases for function expressions with type annotations, we have to make a design decision. We can either keep our old typing rules, which require the programmer to annotate functions with the exact return type of the function body, or we can relax the condition and only require that the annotated type is a supertype of the actual type of the function body. We choose the latter because it provides a useful mechanism for information hiding. For instance, consider the following typed variant of our counter class with open recursion from class 21:

```javascript
class counterClass =
function(this: { x: number; get: () => number; inc: () => () })
: { get () => number; inc () => () }
{
  this.x = 0;
  this.get = function() { return this.x; };
  this.inc = function() { this.x = this.x + 1; }
  return this;
};
```

The annotated return type for the function `counterClass` is a supertype of the actual type of the returned expression `this`. The effect is that the return type hides the field `x` of `this`, which we use for the representation of the internal state of the counter object. A client that uses an instance of the counter class can only access the field `x` by calling the methods `get` and `inc`.

Assignment Expressions. The rules `TypeAssignVar` and `TypeAssignFld` are the new rules for typing assignment expressions. Note that the rules require that the inferred type `τ′` of the assigned location expression is a supertype of the type `τ` inferred for the right side of the assignment. Thus, location expressions are typed contravariantly. These modified rules plug the loop hole in our previous type system with the unrestricted subtyping rule `TypeSub`. In particular, our problematic program from previous class is now rejected by the new typing relation:

```javascript
const x = {f: {g: true}};
x.f = {};
x.f.g
```

The assignment expression on line 2 is ill-typed because the type `{}` is not a subtype of the type `{var g: bool}`.
Conditional Expressions. Perhaps the most interesting new typing rule is the rule TypeIf for conditional expressions $e_1 ? e_2 : e_3$. In our earlier version of this rule, we required that the inferred types of the “then” branch $e_2$ and the “else” branch $e_3$ must agree. With the subtyping rule we can relax the inferred types of the two branches to a common supertype. For example, with subtyping the following program is well-typed:

```plaintext
const o = true ? {x: 0, y: true} : {x: 2, z: 0};
2 * o.x
```

The inferred type of the “then” branch ${x: 0, y: true}$ is

$$\{\text{var } x: \text{number}; \text{var } y: \text{bool}\}$$

and the inferred type of the “else” branch ${x: 2, z: 0}$ is

$$\{\text{var } x: \text{number}; \text{var } z: \text{number}\}$$

While the two types do not agree, they have a common supertype

$$\{\text{var } x: \text{number}\}$$

Thus, we can apply the rule TYPESUB to each branch, relaxing the inferred type to the common supertype. We can then successfully type the conditional expression with the old TypeIf rule. Moreover, the common supertype is sufficient to successfully type the body of the const declaration, since the expression $2 * o.x$ only accesses the field $x$ of $o$, which it expects to be of type number. Hence, the above program is well-typed.

How do we incorporate the rule TYPESUB directly into the rule TypeIf? We want to be able to relax the inferred types of the two branches to a common supertype, but which supertype should we pick? For example, in the above program another common supertype is the empty object type `{}`. This supertype could also be used to merge the two branches in the conditional. Hence, the conditional expression would still be well-typed. However, we would then fail to successfully type the subexpression $o.x$ in the body of the const declaration if the inferred type for $o$ was the empty object type `{}`. When we merge the inferred types of the two branches, we have to preserve as much information as possible that is common to the two types. More precisely, the type of a conditional expression should be the least common supertype of the types inferred for its two branches.

We refer to the least common supertype of two types $\tau_1$ and $\tau_2$ as the join of the two types, which we denote by $\tau_1 \sqcup \tau_2$. The intuition is that we find $\tau_1 \sqcup \tau_2$ by walking up the lattice of the subtyping relation from both $\tau_1$ and $\tau_2$ until we find the first common supertype. For example, we have

$$\{\text{var } x: \text{number}; \text{var } y: \text{bool}\} \sqcup \{\text{var } x: \text{number}; \text{var } z: \text{number}\} = \{\text{var } x: \text{number}\}$$

Hence, the type that the new TypeIf rule infers for the expression

$$true ? \{x: 0, y: true\} : \{x: 2, z: 0\}$$
is \{\texttt{var} \ x:\texttt{number}\}. You may ask yourself why the inferred type is not

\{\texttt{var} \ x:\texttt{number} \ \texttt{var} \ y:\texttt{bool}\}

since we know that the above conditional expression will always take the “then” branch. The reason is that the typing rules must consider the general case. In general, we cannot predict the truth value of branching conditions statically without evaluating the program. The typing relation therefore always assumes that both branches could be taken and infers the type that captures the maximal information common to the types of both branches. This type is obtained by taking the join of the types that are inferred for the two branches.

In our subtype relation, joins do not always exist. For example, the two basic types \texttt{number} and \texttt{bool} do not have any common supertype. Hence, the following program will be considered ill-typed because the rule TYPEIF does not apply:

\texttt{true \ ? \ 3 : false}

We discuss below how the computation of joins actually works.

\textbf{Equality Expressions.} Finally, the TYPEEQUAL rule is a slightly relaxed version of our earlier typing rule for equality and disequality expressions. Instead of requiring that we only compare expressions \(e_1\) and \(e_2\) of the same type, we now require that the inferred types \(\tau_1\) and \(\tau_2\) of the two expressions must have some common supertype. We express this condition by stating that the join \(\tau_1 \sqcup \tau_2\) must exist.

\textbf{Computing Joins and Meets}

\textbf{Joins.} A join of two types \(\tau_1\) and \(\tau_2\) is a type \(\tau\) that satisfies the following two properties: (1) \(\tau\) is a supertype of both \(\tau_1\) and \(\tau_2\) and (2) any other supertype \(\tau'\) of \(\tau_1\) and \(\tau_2\) must also be a supertype of \(\tau\).

In our type system, joins are not unique. That is, given \(\tau_1\) and \(\tau_2\), we can have two distinct types \(\tau\) and \(\tau'\) such that both satisfy the conditions (1) and (2) above. For example, consider the types:

\[
\tau_1 = \{x: \texttt{number}; y: \texttt{bool}; z: \texttt{bool}\}
\]

\[
\tau_2 = \{x: \texttt{number}; y: \texttt{number}; z: \texttt{bool}\}
\]

Then both of the following types are joins of \(\tau_1\) and \(\tau_2\):

\[
\tau = \{x: \texttt{number}; z: \texttt{bool}\}
\]

\[
\tau' = \{z: \texttt{bool}; x: \texttt{number}\}
\]

Note that we have \(\tau <: \tau'\) and \(\tau' <: \tau\). In fact, \(\tau\) and \(\tau'\) are equal up to permutation of their fields. This is true in general: any two joins of two types \(\tau_1\) and \(\tau_2\) will be equal up to permutation of fields in their object type subexpressions.
In our discussion, we will treat these different joins as equal and simply write \( \tau_1 \sqcup \tau_2 \) for the join of \( \tau_1 \) and \( \tau_2 \).\(^1\)

If the join of two types \( \tau_1 \) and \( \tau_2 \) does not exist, we indicate this by writing \( \tau_1 \sqcup \tau_2 = \bot \). That is, we complete \( \sqcup \) to a total function on pairs of types.

The inference rules in Figure 3 describe how the function \( \sqcup \) can be computed using recursion on the structure of type expressions. We have omitted the rules that propagate the proof of non-existence of a join, indicated by \( \bot \), from subderivations to the top-level. We discuss the individual rules in more detail below.

**Joins of Basic Types.** If one of the two types \( \tau_1 \) and \( \tau_2 \) is a basic type, then computing their join \( \tau_1 \sqcup \tau_2 \) simply reduces to checking whether the two types are equal or not. This is because basic types have no proper supertypes. The corresponding case analysis is captured by the rules JoinBasic, JoinBasicFail, and JoinBasicFail2.

**Joins of Object Types.** Next, we discuss the case where \( \tau_1 \) and \( \tau_2 \) are both object types. The join computation for object types is handled by the JoinObj rules. First, note that in this case the join always exists because \( \tau_1 \) and \( \tau_2 \) have at least one common supertype, namely, the empty object type \( \{\} \). Intuitively, we obtain the actual join \( \tau \) by taking all fields \( h \) that are common to both \( \tau_1 \) and \( \tau_2 \), and merging the types of \( h \) appropriately. If \( h \) is a \texttt{const} field in both object types, then merging means that we compute the join recursively to reflect the depth subtyping of such fields. In the case where \( h \) is a \texttt{var} field in both types, we must require that \( h \) has the same type in \( \tau_1 \) and \( \tau_2 \) for \( h \) to be included in the join \( \tau \). In all other cases, \( h \) will not be included in \( \tau \) (i.e., if \( h \) is a \texttt{const} field whose recursive join does not exist, or if \( h \) has different mutabilities in \( \tau_1 \) and \( \tau_2 \)). The join computation works by recursion over the fields of \( \tau_1 \). The base case is captured by the rule JoinObjEmp. The remaining rules capture the cases where \( \tau_1 \) has at least one field. The recursion then goes over the left-most field \( h \) of \( \tau_1 \) by case splitting on whether \( h \) is a \texttt{const} or a \texttt{var} field, whether it occurs in \( \tau_2 \), etc.

**Joins of Function Types.** Finally, we consider the case of computing the join of two function types \((x : \tau_1) \Rightarrow \tau_2\) and \((y : \tau_1') \Rightarrow \tau_2'\). If the join exists, it is again a function type of the form \((x : \tau_1'') \Rightarrow \tau_2''\). Though, how do we compute the types \( \tau_1'' \) and \( \tau_2'' \)? Since function types are covariant in their result types, it is easy to see that we must have \( \tau_2'' = \tau_2 \sqcup \tau_2' \). The following program exemplifies this intuition:

\(^1\)More formally, we consider the quotient of \( \text{Typ} \) under the equivalence relation that is induced by the subtype relation. Then \( \tau_1 \sqcup \tau_2 \) is the least upper bound of the equivalence classes represented by \( \tau_1 \) and \( \tau_2 \) in the partial order that is induced by \( < \) on this quotient. The least upper bounds of a partial order are always unique if they exist.
\[\tau \in \{\text{bool, number}\} \quad \text{JOINBasic}\]

\[\tau_1 \neq \tau_2 \quad \tau_1 \in \{\text{bool, number}\} \quad \text{JOINBasicFail}_1\]

\[\tau_1 \neq \tau_2 \quad \tau_2 \in \{\text{bool, number}\} \quad \text{JOINBasicFail}_2\]

\[\{\} \cup \{\text{mut}_g g : \tau_g\} = \{\}\quad \text{JOINObjEmp}\]

\[\{\text{mut}_f f : \tau_f\} \cup \{\text{mut}_g g : \tau_g; \text{mut}_g' g' : \tau_g'\} = \{\text{mut}_k k : \tau_k\}\]
\[\tau_1 \cup \tau_2 = \top \quad \text{JOINObjC}_1\]

\[\{\text{const} h : \tau_h; \text{mut}_f f : \tau_f\} \cup \{\text{mut}_g g : \tau_g; \text{mut}_g' g' : \tau_g'\} = \tau\]
\[\{\text{const} h : \tau_h; \text{mut}_f f : \tau_f\} \cup \{\text{mut}_g g : \tau_g; \text{mut}_g' g' : \tau_g'\} = \tau\]
\[\text{JOINObjC}_2\]

\[\{\text{var} h : \tau_h; \text{mut}_f f : \tau_f\} \cup \{\text{mut}_g g : \tau_g; \text{mut}_g' g' : \tau_g'\} = \tau\]
\[\text{JOINObjV}_1\]

\[\tau_1 \neq \tau_2 \quad \{\text{mut}_f f : \tau_f\} \cup \{\text{mut}_g g : \tau_g; \text{mut}_g' g' : \tau_g'\} = \tau\]
\[\text{JOINObjV}_2\]

\[\text{JOINObjNE}\]

\[h \notin \{\\} \quad \{\text{mut}_f f : \tau_f\} \cup \{\text{mut}_g g : \tau_g\} = \tau\quad \text{JOINObjNo}\]

\[\{\text{mut}_1 h : \tau_1; \text{mut}_f f : \tau_f\} \cup \{\text{mut}_g g : \tau_g\} = \tau\]

\[\tau_1 \cap \tau'_1 = \tau'_2 \quad \tau_2 \cup \tau'_2 = \tau''_2 \quad (x : \tau'_1) \Rightarrow \tau'_2 \quad (x : \tau_1) \Rightarrow \tau_2 \quad \text{JOINFun}\]

Figure 3: Rules for computing joins. The rules for propagating the non-existence of a join have been elided.
const f1 = function () { return { x: 0, y: true }; }
const f2 = function () { return { x: 1, z: 1 } };
const f = true ? f1 : f2;
f().x

The inferred type for f must be such that all subsequent usages of f are safe, regardless of whether f is evaluated to f1 or f2. The expression f().x accesses the field x of f’s return value. This is safe since the objects returned by f1 and f2 both have a field x of type number. However, if we replaced f().x by f().z, then the program would only be safe if the conditional expression defining f always took the “else” branch. The inferred return type for f should capture the maximal information that is common to the return types of f1 and f2, i.e., we have to compute the join of the two return types.

For the parameter types of two joined function types the situation is reversed since parameter types are contravariant. Consider the following example:

const f1 = function (o: { x: number; y: number }) { return o.x + o.y }; const f2 = function (o: { x: number; z: number }) { return o.x + o.z }; const f = true ? f1 : f2;
f({x: 1, y: 2, z: 3})

The inferred type of f1 is

(o:{var x:number;var y:number}) ⇒ number

and the inferred type of f2 is

(o:{var x:number;var z:number}) ⇒ number

The call to f must be safe regardless of whether f is bound to f1 or f2. Note that f1 accesses fields x and y of the passed object whereas f2 accesses fields x and z. Hence, the object that is passed to f must at least have the fields x, y, and z. Computing the join of the parameter types of f1 and f2 yields

{var x:number}

Thus, if we defined the parameter type of the join of the types of f1 and f2 to be {var x:number}, then the call f({x: 1}) would be well-typed, even though it is unsafe.

Instead of computing the type that captures the maximal information that is common to both parameter types, we have to compute the type that captures all the information provided in each of the two types. This means that we have to compute their greatest common subtype. We refer to the greatest common subtype of two types \( \tau_1 \) and \( \tau_2 \) as their meet, which we denote by \( \tau_1 \cap \tau_2 \).
The computation of the join of function types is then summarized by the rule **JoinFun**.

Due to the contravariant subtyping of parameter types in function types, we have to compute joins and meets simultaneously using mutual recursion. Whenever we compute the join of two function types, we switch to computing the meets of their parameter types. Similarly, whenever we compute the meet of two function types, we switch back to computing the joins of their parameter types. This mutual recursion is well-founded since we always recurse into smaller subexpressions of the considered types. We next explain the computation of meets in detail.

**Meets.** As indicated in our discussion above, the meet of two types is the order-theoretic dual of their join\(^2\). Similar to joins, meets are unique up to reordering of fields in object type expressions. So we denote by \(\tau_1 \sqcap \tau_2\) one of the representatives of all these equivalent meets if they exist. If no meet exists, we indicate this fact by writing \(\tau_1 \sqcap \tau_2 = \bot\) as in the case of joins. Figure 4 provides inference rules for computing meets using structural recursion on the type expressions. The rules are very similar to the rules for joins. In particular, the rule for computing meets of function types, **MeetFun**, is the dual of the rule **JoinFun**. That is, we obtain **MeetFun** from **JoinFun** by replacing meets by joins and vice versa. In the following, we only discuss the cases for computing meets of object types in more detail.

**Meets of Object Types.** In order to compute the meet \(\tau = \tau_1 \sqcap \tau_2\) of two object types \(\tau_1\) and \(\tau_2\), we proceed as follows. First, we collect all fields that occur in one of \(\tau_1\) and \(\tau_2\) but not the other and add them to \(\tau\) with their respective mutabilities and types. For each field \(h\) that is common to \(\tau_1\) and \(\tau_2\), we first check whether it has the same mutability in both types. If not, the meet \(\tau\) does not exist. If the mutability is consistent, we merge \(h\)'s types in \(\tau_1\) and \(\tau_2\) similar to the computation of joins, except that if we merge the types of a **const** field, we now take their meets instead of their joins. The individual cases are captured by the **MeetObj** rules. Again, the computation recurses over the left-most field of type \(\tau_1\).

Note that if \(h\) is a **const** field in both object types, but the meet of the two types of \(h\) does not exist, then the join of the two object types themselves does not exist. The corresponding case is handled by one of the propagation rules for \(\bot\), which have been elided in Figure 4. Some languages include null pointers, which can be thought of as belonging to a type **Null** that is a common subtype of all object types. In such languages, the meet of two object types always exists. If we included such a type **Null** in our type system, we would replace the propagation rule described above by the following rule:

\[
\begin{align*}
\tau_1 \sqcap \tau_2 = \bot & \quad \text{MeetObjC}_2 \\
\{\text{const } h : \tau_1; \text{mut } f : \tau_f\} \sqcap \{\text{mut } g : \tau_g; \text{const } h : \tau_2; \text{mut } g' : \tau_{g'}\} = \text{Null}
\end{align*}
\]

Similarly, the rule **MeetObjV** would return **Null** instead of \(\bot\).

\(^2\)That is, meets are the greatest lower bounds of the partial order that is induced by the subtype relation.
\[
\begin{align*}
\tau &\in \{\text{bool, number}\} \quad \text{MeetBasic} \\
\tau_1 \neq \tau_2 \quad \tau_1 \in \{\text{bool, number}\} \quad \text{MeetBasicFail}_1 \\
\tau_1 \neq \tau_2 \quad \tau_2 \in \{\text{bool, number}\} \quad \text{MeetBasicFail}_2 \\
\emptyset \sqcap \{\text{mut}_g g : \tau_g\} &= \{\text{mut}_g g : \tau_g\} \quad \text{MeetObjEmp} \\
\{\text{mut}_f f : \tau_f\} \sqcap \{\text{mut}_g g : \tau_g; \text{mut}_{g'} g' : \tau_{g'}\} &= \{\text{mut}_k k : \tau_k\} \quad \text{MeetObjC} \\
\text{const} h : \tau_h; \text{mut}_k k : \tau_k \quad \tau &= \{\text{const} h : \tau_h; \text{mut}_k k : \tau_k\} \quad \text{MeetObjV}_1 \\
\text{var} h : \tau_h; \text{mut}_f f : \tau_f \quad \text{var} h : \tau_h; \text{mut}_{g'} g' : \tau_{g'} \quad \tau &= \{\text{var} h : \tau_h; \text{mut}_{g'} g' : \tau_{g'}\} \quad \text{MeetObjV}_2 \\
\text{mut}_1 \neq \text{mut}_2 \quad \{\text{mut}_1 h : \tau_1; \text{mut}_f f : \tau_f\} \sqcap \{\text{mut}_g g : \tau_g; \text{mut}_2 h : \tau_2; \text{mut}_{g'} g' : \tau_{g'}\} &= \emptyset \quad \text{MeetObjNe} \\
\text{mut}_1 \neq \text{mut}_2 \quad \{\text{mut}_f f : \tau_f\} \sqcap \{\text{mut}_g g : \tau_g\} &= \{\text{mut}_k k : \tau_k\} \\
\text{mut}_1 h : \tau_1; \text{mut}_f f : \tau_f \quad \{\text{mut}_g g : \tau_g\} &= \{\text{mut}_k k : \tau_k\} \quad \text{MeetObjNo} \\
\tau_1 \sqcup \tau'_1 \quad \tau_2 \sqcup \tau'_2 \quad \tau'' &= (x : \tau_1') \Rightarrow \tau'_2 \Rightarrow (y : \tau_2') \Rightarrow \tau'' \quad \text{MeetFun}
\end{align*}
\]

Figure 4: Rules for computing meets. The rules for propagating the non-existence of a meet have been elided.