Homework 12

The primary purpose of this assignment is to explore subtyping in object-oriented languages. Concretely, we will extend the type inference algorithm JakartaScript with support for subtyping. Our interpreter will now support most of the core features of modern programming languages.

Try to make your code as concise and clear as possible. Challenge yourself to find the most crisp, concise way of expressing the intended computation. This may mean using ways of expressing computation currently unfamilar to you.

Problem 1 Objects and Subtyping (14 Points)

This is a warm-up exercise to get yourself familiar with subtyping and the computation of joins and meets in the subtype lattice.

(a) Indicate whether the following JakartaScript subtype relationships are true or false.

(i) `Bool <: Num`

(ii) `{let f: Num} <: {let f: Any}`

(iii) `{let f: Num} <: {const f: Any}`

(iv) `{const f: Num} <: {let f: Num}`

(v) `{let f: Num} <: {let g: Num}`

(vi) `{const f: {}} <: {const f: Any, let g: Bool}`

(vii) `Any => Bool <: Bool => Any`

(viii) `Bool => Bool <: Any => Any`

(b) For each of the following programs, determine whether the program is well-typed with subtyping. If it is well-typed, give the type that is inferred for the whole program. If it is not well-typed, provide a brief explanation of the type error.

(i)

```
const x = {let f: 3};
const fun = function(y: {let f: Num, const g: Bool}) {
    y.f = 4; return y;
  };
fun(x).f
```

(ii)

```
const x = {let f: 3, const g: true};
const fun = function(y: {let f: Num}) {
    return y;
  };
fun(x).f
```

(iii)

```
const x = {let f: 3, const g: true};
const fun = function(y: {let f: Num}) {
    return y;
  };
fun(x).g
```

(iv)

```
  const x = {let f: 1, const g: true};
  const y = {const f: false, let g: 2};
  const z = true ? x : y;
  z.f
```

(c) For each of the following pairs of types τ_1 and τ_2, compute their join $\tau_1 \sqcup \tau_2$ and meet $\tau_1 \sqcap \tau_2$.

(i) $\tau_1 =$ **Num**, $\tau_2 =$ {**const** f: **Num**}

(ii) $\tau_1 =$ {}, $\tau_2 =$ {**let** f: **Num**, **const** g: **Bool**}

(iii) $\tau_1 =$ {**let** f: **Num**}, $\tau_2 =$ {**const** g: **Bool**}

(iv) $\tau_1 =$ {**let** f: **Num**, **const** g: {**let** h: **Any**}},
$\tau_2 =$ {**let** f: **Num**, **const** g: {**const** h: **Bool**}}

(v) $\tau_1 =$ (**Any** => **Bool**), $\tau_2 =$ (**Bool** => **Any**)

(vi) $\tau_1 =$ (**Bool** => **Num**), $\tau_2 =$ (**Num** => **Bool**)

Problem 2 JakartaScript Interpreter with Subtyping (26 Points)

Compared to Homework 11, the only extension to JakartaScript that we consider in this final Homework assignment is the addition of the greatest supertype **Any** and the least subtype **Nothing**. The syntax of the new language is shown in Figure 1. In Figure 2, we show the updated and new AST nodes.

Type Checking. The inference rules defining the typing relation are given in figures 3 and 4. The only change compared to Homework 11 are the updated rules that involve subtyping, which are summarized in Figure 4. A template for the new type inference function

```
def typeInfer(env: Map[String,(Mut,Typ)], e: Expr): Typ
```

has been provided for you. The only missing cases are for the new rules in Figure 4. One of your tasks is to implement those missing cases.

Subtyping. The typing rules in Figure 4 make use of the subtype relation $\tau_1 <: \tau_2$. The inference rules defining the subtype relation are given in Figure 5. In our interpreter implementation, this relation is implemented by the Scala function

```
def subtype(t1: Typ, t2: Typ): Boolean
```

$$
\begin{array}{rll}
n \in \mathit{Num} & & \text{numbers (double)} \\
s \in \mathit{Str} & & \text{strings} \\
a \in \mathit{Addr} & & \text{addresses} \\
b \in \mathit{Bool} & ::= \texttt{true} \mid \texttt{false} & \text{Booleans} \\
x \in \mathit{Var} & & \text{variables} \\
f \in \mathit{Fld} & & \text{field names} \\
\tau \in \mathit{Typ} & ::= \texttt{Bool} \mid \texttt{Num} \mid \texttt{String} \mid \texttt{Undefined} \mid & \text{types} \\
 & \quad (\overline{\tau}) \Rightarrow \tau_0 \mid \{\overline{\mathit{mut}\, f : \tau}\} \mid \texttt{Any} \mid \texttt{Nothing} & \\
v \in \mathit{Val} & ::= \texttt{undefined} \mid n \mid b \mid s \mid a \mid \texttt{function}\ p(\overline{x:\tau})\, t\ e & \text{values} \\
e \in \mathit{Expr} & ::= x \mid v \mid \mathit{uop}\ e \mid e_1\ \mathit{bop}\ e_2 \mid e_1\ \texttt{?}\ e_2\ \texttt{:}\ e_3 \mid e.f \mid \{\overline{\mathit{mut}\, f : e}\} & \text{expressions} \\
 & \quad \texttt{console.log}(e) \mid e_1(\overline{e}) \mid \mathit{mut}\ x = e_1\,; e_2 & \\
\mathit{uop} \in \mathit{Uop} & ::= \texttt{-} \mid \texttt{!} \mid \texttt{*} & \text{unary operators} \\
\mathit{bop} \in \mathit{Bop} & ::= \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{===} \mid \texttt{!==} \mid \texttt{<} \mid \texttt{>} \mid & \text{binary operators} \\
 & \quad \texttt{<=} \mid \texttt{>=} \mid \texttt{\&\&} \mid \texttt{||} \mid \texttt{,} \mid \texttt{|=} & \\
p & ::= x \mid \epsilon & \text{function names} \\
t & ::= \texttt{:}\tau \mid \epsilon & \text{return types} \\
\mathit{mut} \in \mathit{Mut} & ::= \texttt{const} \mid \texttt{let} & \text{mutability} \\
k \in \mathit{Con} & ::= v \mid \{\overline{f : v}\} & \text{memory contents} \\
M \in \mathit{Mem} & = \mathit{Addr} \rightharpoonup \mathit{Con} & \text{memories}
\end{array}
$$

Figure 1: Abstract syntax of JakartaScript

We have already implemented the cases for the rules SubRefl and SubAny. Your task is to complete the missing cases for the rules SubNothing, SubObj, and SubFun. For the rule SubObj, we have provided a template. The idea of the implementation of this rule is to iterate over the fields g_j of the right type τ_2 and to attempt a look-up of this field in the map that holds the fields of type τ_1. If the look-up succeeds, then you still need to check the remaining conditions given in rule SubObj.

Joins and Meets. The typing rule for conditional expressions TypeIf and the rule for (dis)equality expressions TypeEqual make use of the join operator $\tau_1 \sqcup \tau_2$, which computes the least common supertype of the types τ_1 and τ_2. The rules defining the join operator are given in Figure 6. Note that the rules only partially define $\tau_1 \sqcup \tau_2$. Our convention is that if there is no type τ such that we can derive $\tau_1 \sqcap \tau_2 = \tau$ with the rules in Figure 7, then we define $\tau_1 \sqcap \tau_2 = \texttt{Any}$. In our interpreter implementation, the join operator is computed by the Scala function

```
def join(t1: Typ, t2: Typ): Typ
```

Again, we have already provided some of the cases for you. You need to complete the

```
...
/** Types */
enum Typ:
  ...
  case TAny, TNothing // <~ Any, Nothing
```

Figure 2: The abstract syntax of JakartaScript as represented in Scala. After each **case**, we show the corresponding JavaScript expression represented by that case.

missing cases for joins of object and function types. Recall that the computation of the join of two function types requires the computation of *meets* on their argument types. The meet operator $\tau_1 \sqcap \tau_2$ is defined by the rules in Figure 7. Dually to the join, for the missing cases we define $\tau_1 \sqcap \tau_2 = \textbf{Nothing}$. In our implementation, the meet operator is computed by the Scala function

```
def meet(t1: Typ, t2: Typ): Typ
```

which you also need to complete.

Evaluation. Compared to Homework 11, we did not add any new features to our language other than the new types **Any** and **Nothing**. The operational semantics of the language is unaffected by adding these new type, which means that we can reuse the implementation of the `eval` function from Homework 11 without modifications. The code package of this homework assignment provides the same template for the `eval` function that we provided for Homework 11. You can simply replace this template with your `eval` function from the previous homework assignment, or with the reference implementation of the `eval` function that will be provided with the sample solution of Homework 11.

$$\frac{}{\Gamma \vdash b : \texttt{Bool}}\ \textsc{TypeBool} \qquad \frac{}{\Gamma \vdash n : \texttt{Num}}\ \textsc{TypeNum} \qquad \frac{}{\Gamma \vdash s : \texttt{String}}\ \textsc{TypeStr}$$

$$\frac{}{\Gamma \vdash \texttt{undefined} : \texttt{Undefined}}\ \textsc{TypeUndefined}$$

$$\frac{\Gamma \vdash e : \texttt{Num}}{\Gamma \vdash \texttt{-}\, e : \texttt{Num}}\ \textsc{TypeUMinus} \qquad \frac{\Gamma \vdash e : \texttt{Bool}}{\Gamma \vdash \texttt{!}\, e : \texttt{Bool}}\ \textsc{TypeNot}$$

$$\frac{\Gamma \vdash e_1 : \texttt{Bool} \quad \Gamma \vdash e_2 : \texttt{Bool} \quad bop \in \{\texttt{\&\&}, \texttt{||}\}}{\Gamma \vdash e_1\, bop\, e_2 : \texttt{Bool}}\ \textsc{TypeAndOr}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 \texttt{,}\, e_2 : \tau_2}\ \textsc{TypeSeq} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{console.log(}e\texttt{)} : \texttt{Undefined}}\ \textsc{TypePrint}$$

$$\frac{\Gamma \vdash e_1 : \texttt{String} \quad \Gamma \vdash e_2 : \texttt{String}}{\Gamma \vdash e_1 \texttt{+} e_2 : \texttt{String}}\ \textsc{TypePlusStr}$$

$$\frac{\Gamma \vdash e_1 : \texttt{Num} \quad \Gamma \vdash e_2 : \texttt{Num} \quad bop \in \{\texttt{+}, \texttt{*}, \texttt{/}, \texttt{-}\}}{\Gamma \vdash e_1\, bop\, e_2 : \texttt{Num}}\ \textsc{TypeArith}$$

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau \quad \tau \in \{\texttt{Num}, \texttt{String}\} \quad bop \in \{\texttt{>}, \texttt{>=}, \texttt{<}, \texttt{<=}\}}{\Gamma \vdash e_1\, bop\, e_2 : \texttt{Bool}}\ \textsc{TypeInequal}$$

$$\frac{\Gamma \vdash e_d : \tau_d \quad \Gamma' = \Gamma[x \mapsto (mut, \tau_d)] \quad \Gamma' \vdash e_b : \tau_b}{\Gamma \vdash mut\ x \texttt{=} e_d \texttt{;} e_b : \tau_b}\ \textsc{TypeDecl}$$

$$\frac{x \in \mathsf{dom}(\Gamma) \quad \Gamma(x) = (mut, \tau)}{\Gamma \vdash x : \tau}\ \textsc{TypeVar}$$

$$\frac{\begin{array}{c}\Gamma' = \Gamma[x_1 \mapsto (\texttt{const}, \tau_1)] \ldots [x_n \mapsto (\texttt{const}, \tau_n)] \\ \Gamma' \vdash e : \tau \quad \tau' = (\tau_1, \ldots, \tau_n) \Rightarrow \tau\end{array}}{\Gamma \vdash x_1 \texttt{:} \tau_1 \texttt{,} \ldots \texttt{,}\ x_n \texttt{:} \tau_n \texttt{=>}\ e : \tau'}\ \textsc{TypeFun}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \ldots \quad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash \{mut_1\ f_1 \texttt{:} e_1 \texttt{,} \ldots \texttt{,} mut_n\ f_n \texttt{:} e_n\} : \{mut_1\ f_1 \texttt{:} \tau_1 \texttt{,} \ldots \texttt{,} mut_n\ f_n \texttt{:} \tau_n\}}\ \textsc{TypeObj}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : \{mut_1\ f_1 \texttt{:} \tau_1 \texttt{,} \ldots \texttt{,} mut_n\ f_n \texttt{:} \tau_n\} \\ f = f_i \quad \tau = \tau_i \quad i \in [1, n]\end{array}}{\Gamma \vdash e\texttt{.}f : \tau}\ \textsc{TypeDerefFld}$$

Figure 3: Type checking rules for non-object primitives of JakartaScript (no changes compared to Homework 11)

$$\frac{\Gamma \vdash e : (\tau_1, \ldots, \tau_k) \Rightarrow \tau \quad \text{for all } i \in [1,k]\text{:} \quad \Gamma \vdash e_i : \tau_i' \text{ and } \tau_i' <: \tau_i}{\Gamma \vdash e(e_1, \ldots, e_k) : \tau} \text{ TypeCall}$$

$$\frac{\Gamma' = \Gamma[x_1 \mapsto (\texttt{const}, \tau_1)] \ldots [x_k \mapsto (\texttt{const}, \tau_k)] \quad \tau = (\tau_1, \ldots, \tau_k) \Rightarrow \tau_0 \quad \Gamma' \vdash e : \tau_0' \quad \tau_0' <: \tau_0}{\Gamma \vdash (x_1 : \tau_1, \ldots, x_k : \tau_k) \texttt{ => } e \ : \tau} \text{ TypeFunAnn}$$

$$\frac{\Gamma' = \Gamma[x \mapsto \tau][x_1 \mapsto (\texttt{const}, \tau_1)] \ldots [x_k \mapsto (\texttt{const}, \tau_k)] \quad \tau = (\tau_1, \ldots, \tau_k) \Rightarrow \tau_0 \quad \Gamma' \vdash e : \tau_0' \quad \tau_0' <: \tau_0}{\Gamma \vdash \texttt{function}\, x(x_1 : \tau_1, \ldots, x_k : \tau_k) : \tau_0\ e \ : \tau} \text{ TypeFunRec}$$

$$\frac{\Gamma(x) = (\texttt{let}, \tau') \quad \Gamma \vdash e : \tau \quad \tau <: \tau'}{\Gamma \vdash x = e : \tau} \text{ TypeAssignVar}$$

$$\frac{\Gamma \vdash e_1 : \{ \ldots \texttt{let}\ f : \tau' \ldots \} \quad \Gamma \vdash e_2 : \tau \quad \tau <: \tau'}{\Gamma \vdash e_1.f = e_2 : \tau} \text{ TypeAssignFld}$$

$$\frac{\Gamma \vdash e_1 : \texttt{Bool} \quad \Gamma \vdash e_2 : \tau_2 \quad \Gamma \vdash e_3 : \tau_3 \quad \tau_2 \sqcup \tau_3 = \tau}{\Gamma \vdash e_1 \texttt{ ? } e_2 \texttt{ : } e_3 : \tau} \text{ TypeIf}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_1 \sqcup \tau_2 \neq \texttt{Any} \quad \tau_1, \tau_2 \text{ have no function types} \quad \mathit{bop} \in \{\texttt{===}, \texttt{!==}\}}{\Gamma \vdash e_1\ \mathit{bop}\ e_2 : \texttt{Bool}} \text{ TypeEqual}$$

Figure 4: Type checking rules of JakartaScript that involve subtyping

$$\frac{}{\texttt{Nothing} <: \tau} \text{ SubNothing} \qquad \frac{}{\tau <: \texttt{Any}} \text{ SubAny} \qquad \frac{}{\tau <: \tau} \text{ SubRefl}$$

$$\frac{\tau <: \tau' \qquad \text{for all } i \in [1,k]\text{: } \tau_i' <: \tau_i}{((\tau_1, \ldots, \tau_k) \Rightarrow \tau) <: ((\tau_1', \ldots, \tau_k') \Rightarrow \tau')} \text{ SubFun}$$

$$\frac{\{g_1, \ldots, g_m\} \subseteq \{f_1, \ldots, f_k\} \quad \text{for all } i, j, \text{ if } f_i = g_j, \text{ then } \mathit{mut}_j' = \texttt{const} \text{ and } \tau_i <: \tau_j' \text{ or } \mathit{mut}_i = \mathit{mut}_j' \text{ and } \tau_i = \tau_j'}{\{\mathit{mut}_1\ f_1 : \tau_1, \ldots ; \mathit{mut}_k\ f_k : \tau_k\} <: \{\mathit{mut}_1'\ g_1 : \tau_1' ; \ldots, \mathit{mut}_m'\ g_m : \tau_m'\}} \text{ SubObj}$$

Figure 5: Subtyping rules of JakartaScript

$$\frac{}{\texttt{Nothing} \sqcup \tau = \tau}\ \textsc{JoinNothing}_1 \qquad \frac{}{\tau \sqcup \texttt{Nothing} = \tau}\ \textsc{JoinNothing}_2$$

$$\frac{\tau \in \{\texttt{Bool}, \texttt{Num}, \texttt{String}, \texttt{Undefined}\}}{\tau \sqcup \tau = \tau}\ \textsc{JoinBasic}_=$$

$$\frac{}{\{\} \sqcup \{mut_g\ g{:}\tau_g\} = \{\}}\ \textsc{JoinObjEmp}$$

$$\frac{h \notin \{\overline{g}\} \quad \{\overline{mut_f\ f{:}\tau_f}\} \sqcup \{\overline{mut_g\ g{:}\tau_g}\} = \tau''}{\{mut_h\ h{:}\tau_h, \overline{mut_f\ f{:}\tau_f}\} \sqcup \{\overline{mut_g\ g{:}\tau_g}\} = \tau''}\ \textsc{JoinObjNo}$$

$$\frac{\begin{array}{c}(\tau_f \neq \tau'_f \text{ or } mut_f \neq mut'_f) \quad \{\overline{mut_g\ g{:}\tau_g}\} \sqcup \{\overline{mut_{g'}\ g'{:}\tau_{g'}}\} = \{\overline{mut_h\ h{:}\tau_h}\} \\ \tau_f \sqcup \tau'_f = \tau''_f \quad \tau = \{\texttt{const}\ f{:}\tau''_f, \overline{mut_h\ h{:}\tau_h}\}\end{array}}{\{mut_f\ f{:}\tau_f, \overline{mut_g\ g{:}\tau_g}\} \sqcup \{mut'_f\ f{:}\tau'_f, \overline{mut_{g'}\ g'{:}\tau_{g'}}\} = \tau}\ \textsc{JoinObjMut}_{\neq}$$

$$\frac{\begin{array}{c}\{\overline{mut_g\ g{:}\tau_g}\} \sqcup \{\overline{\overline{mut_{g'}\ g'{:}\tau_{g'}}}\} = \{\overline{mut_h\ h{:}\tau_h}\} \\ \tau = \{mut_f\ f{:}\tau_f, \overline{mut_h\ h{:}\tau_h}\}\end{array}}{\{mut_f\ {:}\tau_f, \overline{mut_g\ f{:}\tau_g}\} \sqcup \{mut_f\ f{:}\tau_f, \overline{mut_{g'}\ f'{:}\tau_{g'}}\} = \tau}\ \textsc{JoinObjMut}_=$$

$$\frac{\begin{array}{c}\text{for all } i \in [1,k]\text{: } \tau_i \sqcap \tau'_i = \tau''_i \\ \tau_0 \sqcup \tau'_0 = \tau''_0 \quad \tau'' = (\tau''_1, \ldots \tau''_k) \Rightarrow \tau''_0\end{array}}{((\tau_1, \ldots, \tau_k) \Rightarrow \tau_0) \sqcup ((\tau'_1, \ldots, \tau'_k) \Rightarrow \tau'_0) = \tau''}\ \textsc{JoinFunMeet}$$

Figure 6: Rules for computing joins

$$\frac{}{\mathsf{Any} \sqcap \tau = \tau}\ \textsc{MeetAny}_1 \qquad \frac{}{\tau \sqcap \mathsf{Any} = \tau}\ \textsc{MeetAny}_2$$

$$\frac{\tau \in \{\mathsf{Bool}, \mathsf{Num}, \mathsf{String}, \mathsf{Undefined}\}}{\tau \sqcap \tau = \tau}\ \textsc{MeetBasic}_=$$

$$\frac{}{\{\} \sqcap \{mut_g\ g{:}\tau_g\} = \{mut_g\ g{:}\tau_g\}}\ \textsc{MeetObjEmp}$$

$$\frac{\begin{array}{c}\{\overline{mut_g\ g{:}\tau_g}\} \sqcap \{\overline{mut_{g'}\ g'{:}\tau_{g'}}\} = \{\overline{mut_h\ h{:}\tau_h}\} \\ f \notin \{\overline{g'}\} \quad \tau'' = \{mut_f\ f{:}\tau_f, \overline{mut_h\ h{:}\tau_h}\}\end{array}}{\{mut_f\ f{:}\tau_f, \overline{mut_g\ g{:}\tau_g}\} \sqcap \{\overline{mut_{g'}\ g'{:}\tau_{g'}}\} = \tau''}\ \textsc{MeetObjNo}$$

$$\frac{\begin{array}{c}\{\overline{mut_g\ g{:}\tau_g}\} \sqcap \{\overline{mut_{g'}\ g'{:}\tau_{g'}}\} = \{\overline{mut_h\ h{:}\tau_h}\} \\ \tau_f \sqcap \tau'_f = \tau''_f \quad \tau = \{\mathsf{const}\ f{:}\tau''_f, \overline{mut_h\ h{:}\tau_h}\}\end{array}}{\{\mathsf{const}\ f{:}\tau_f, \overline{mut_g\ g{:}\tau_g}\} \sqcap \{\mathsf{const}\ f{:}\tau'_f, \overline{mut_{g'}\ g'{:}\tau_{g'}}\} = \tau}\ \textsc{MeetObjConst}$$

$$\frac{\begin{array}{c}\{\overline{mut_g\ g{:}\tau_g}\} \sqcap \{\overline{mut_{g'}\ g'{:}\tau_{g'}}\} = \{\overline{mut_h\ h{:}\tau_h}\} \\ \tau = \{\mathsf{let}\ f{:}\tau_f, \overline{mut_h\ h{:}\tau_h}\}\end{array}}{\{\mathsf{let}\ f{:}\tau_f, \overline{mut_g\ g{:}\tau_g}\} \sqcap \{\mathsf{let}\ f{:}\tau_f, \overline{mut_{g'}\ g'{:}\tau_{g'}}\} = \tau}\ \textsc{MeetObjLet}$$

$$\frac{\begin{array}{c}\{\overline{mut_g\ g{:}\tau_g}\} \sqcap \{\overline{mut_{g'}\ g'{:}\tau_{g'}}\} = \{\overline{mut_h\ h{:}\tau_h}\} \\ \tau_f <: \tau'_f \quad \tau = \{\mathsf{let}\ f{:}\tau_f, \overline{mut_h\ h{:}\tau_h}\}\end{array}}{\{\mathsf{let}\ f{:}\tau_f, \overline{mut_g\ g{:}\tau_g}\} \sqcap \{\mathsf{const}\ f{:}\tau'_f, \overline{mut_{g'}\ g'{:}\tau_{g'}}\} = \tau}\ \textsc{MeetObjLetConst}$$

$$\frac{\begin{array}{c}\{\overline{mut_g\ g{:}\tau_g}\} \sqcap \{\overline{mut_{g'}\ g'{:}\tau_{g'}}\} = \{\overline{mut_h\ h{:}\tau_h}\} \\ \tau'_f <: \tau_f \quad \tau = \{\mathsf{let}\ f{:}\tau'_f, \overline{mut_h\ h{:}\tau_h}\}\end{array}}{\{\mathsf{const}\ f{:}\tau_f, \overline{mut_g\ g{:}\tau_g}\} \sqcap \{\mathsf{let}\ f{:}\tau'_f, \overline{mut_{g'}\ g'{:}\tau_{g'}}\} = \tau}\ \textsc{MeetObjConstLet}$$

$$\frac{\begin{array}{c}\text{for all } i \in [1,k]\text{: } \tau_i \sqcup \tau'_i = \tau''_i \\ \tau_0 \sqcap \tau'_0 = \tau''_0 \quad \tau'' = (\tau''_1, \ldots \tau''_k) \Rightarrow \tau''_0\end{array}}{((\tau_1, \ldots, \tau_k) \Rightarrow \tau_0) \sqcap ((\tau'_1, \ldots, \tau'_k) \Rightarrow \tau'_0) = \tau''}\ \textsc{MeetFunJoin}$$

Figure 7: Rules for computing meets