

## Homework 12

Due date: Wednesday, December 16, 2015

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.

Like last time, you will work on this assignment in pairs. However, note that each student needs to submit a write-up and are individually responsible for completing the assignment. You are welcome to talk about these exercises in larger groups. However, we ask that you write up your answers in pairs. Also, be sure to acknowledge those with which you discussed, including your partner and those outside of your pair.

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 unfamiliar to you.

Finally, make sure that your file compiles and runs (using Scala 2.11.7). A program that does not compile will *not* be graded.

**Submission instructions.** Upload to NYU classes exactly two files named as follows:

- `hw12-YourNYULogin.pdf` with your answers to the written questions (scanned, clearly legible handwritten write-ups are acceptable).
- `hw12-YourNYULogin.scala` with your answers to the coding exercises.

Replace `YourNYULogin` with your NYU login ID (e.g., I would submit `hw12-tw47.scala` and so forth). To help with managing the submissions, we ask that you rename your uploaded files in this manner.

**Getting started.** Download the code pack `hw12.zip` from the assignment section on the NYU classes page.

**Problem 1 Computing Meets and Joins (12 Points)**

This is a warm-up exercise to make yourself familiar with the join and meet operations on our type language with subtyping.

For each of the following pairs of types  $\tau_1$  and  $\tau_2$ , compute their join  $\tau_1 \sqcup \tau_2$  and meet  $\tau_1 \sqcap \tau_2$ . If the meet does not exist, indicate this by writing  $\tau_1 \sqcap \tau_2 = \perp$ .

- (a)  $\tau_1 = \mathbf{number}$ ,  $\tau_2 = \{\mathbf{const} \ f: \mathbf{number}\}$
- (b)  $\tau_1 = \{\}$ ,  $\tau_2 = \{\mathbf{var} \ f: \mathbf{number}, \mathbf{const} \ g: \mathbf{bool}\}$
- (c)  $\tau_1 = \{\mathbf{var} \ f: \mathbf{number}\}$ ,  $\tau_2 = \{\mathbf{const} \ g: \mathbf{bool}\}$
- (d)  $\tau_1 = \{\mathbf{var} \ f: \mathbf{number}, \mathbf{const} \ g: \{\mathbf{var} \ h: \mathbf{any}\}\}$ ,  
 $\tau_2 = \{\mathbf{var} \ f: \mathbf{number}, \mathbf{const} \ g: \{\mathbf{const} \ h: \mathbf{bool}\}\}$
- (e)  $\tau_1 = (\mathbf{any} \Rightarrow \mathbf{bool})$ ,  $\tau_2 = (\mathbf{bool} \Rightarrow \mathbf{any})$
- (f)  $\tau_1 = (\mathbf{bool} \Rightarrow \mathbf{number})$ ,  $\tau_2 = (\mathbf{number} \Rightarrow \mathbf{bool})$

**Problem 2 JAKARTA SCRIPT Interpreter with Subtyping (28 Points)**

Compared to Homework 11, the only extension to JAKARTA SCRIPT that we consider in this final Homework assignment is the addition of the most general supertype **any**. 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
```

We have already implemented the cases for the rules SUBREFL and SUBANY. Your task is to complete the missing cases for the rules the rules 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  $\tau_2$  and to attempt a look-up of this field in the map that holds the fields of type  $\tau_1$ . If the look-up succeeds, then you still need to check the remaining conditions given in rule SUBOBJ.

$n \in Num$	numbers (double)
$s \in Str$	strings
$a \in Addr$	addresses
$b \in Bool ::= \mathbf{true} \mid \mathbf{false}$	Booleans
$x \in Var$	variables
$f \in Fld$	field names
$\tau \in Typ ::= \mathbf{bool} \mid \mathbf{number} \mid \mathbf{string} \mid \mathbf{Undefined} \mid$ $(\bar{\tau}) \Rightarrow \tau_0 \mid \{\overline{mut f : \tau}\} \mid \mathbf{any}$	types
$v \in Val ::= \mathbf{undefined} \mid n \mid b \mid s \mid a \mid \mathbf{function} \ p(\overline{x : \tau}) \ t \ e$	values
$e \in Expr ::= x \mid v \mid uop \ e \mid e_1 \ bop \ e_2 \mid e_1 \ ? \ e_2 : e_3 \mid e.f \mid \{\overline{mut f : e}\}$ $\mathbf{console.log}(e) \mid e_1(\bar{e}) \mid mut \ x = e_1 ; e_2$	expressions
$uop \in Uop ::= - \mid ! \mid *$	unary operators
$bop \in Bop ::= + \mid - \mid * \mid / \mid == \mid !== \mid < \mid > \mid$ $<= \mid >= \mid \&\& \mid    \mid , \mid =$	binary operators
$p ::= x \mid \epsilon$	function names
$t ::= : \tau \mid \epsilon$	return types
$mut \in Mut ::= \mathbf{const} \mid \mathbf{var}$	mutability
$k \in Con ::= v \mid \{\overline{f : v}\}$	memory contents
$M \in Mem = Addr \rightarrow Con$	memories

Figure 1: Abstract syntax of JAKARTA SCRIPT

**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  $\tau_1$  and  $\tau_2$ . The rules defining the join operator are given in Figure 6. 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 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. In our implementation, the meet operator is computed by the Scala function

```
def meet(t1: Typ, t2: Typ): Option[Typ]
```

which you also need to complete. Note that the meet of two types  $\tau_1$  and  $\tau_2$  does not always exist. In our formal representation, we indicate this case by writing  $\tau_1 \sqcap \tau_2 = \perp$ . The corresponding rules for these cases are omitted in Figure 6. Our convention is that if

```
sealed abstract class Expr extends Positional
...
sealed abstract class Typ
case object TAny extends Typ
TAny any
```

Figure 2: Representing in Scala the abstract syntax of JAKARTA SCRIPT. After each **case class** or **case object**, we show the correspondence between the representation and the concrete syntax.

there is no type  $\tau$  such that we can derive  $\tau_1 \sqcap \tau_2 = \tau$  with the rules in Figure 7, then we define  $\tau_1 \sqcap \tau_2 = \perp$ . In your implementation of `meet`, a call `meet( $\tau_1$ ,  $\tau_2$ )` should return `Some( $\tau$ )` where  $\tau$  is the meet of  $\tau_1$  and  $\tau_2$  if it exists, and `None` if the meet does not exist.

**Evaluation.** Compared to Homework 11, we did not add any new features to our language other than the new type **any**. The operational semantics of the language is unaffected by adding this 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 by 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.

$$\begin{array}{c}
\frac{}{\Gamma \vdash b : \mathbf{bool}} \text{TYPEBOOL} \quad \frac{}{\Gamma \vdash n : \mathbf{number}} \text{TYPERNUM} \quad \frac{}{\Gamma \vdash s : \mathbf{string}} \text{TYPESTR} \\
\\
\frac{}{\Gamma \vdash \mathbf{undefined} : \mathbf{Undefined}} \text{TYPEUNDEFINED} \\
\\
\frac{\Gamma \vdash e : \mathbf{number}}{\Gamma \vdash -e : \mathbf{number}} \text{TYPEUMINUS} \quad \frac{\Gamma \vdash e : \mathbf{bool}}{\Gamma \vdash !e : \mathbf{bool}} \text{TYPENOT} \\
\\
\frac{\Gamma \vdash e_1 : \mathbf{bool} \quad \Gamma \vdash e_2 : \mathbf{bool} \quad bop \in \{\&\&, ||\}}{\Gamma \vdash e_1 \text{ } bop \text{ } e_2 : \mathbf{bool}} \text{TYPEANDOR} \\
\\
\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1, e_2 : \tau_2} \text{TYPESEQ} \quad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathbf{console.log}(e) : \mathbf{Undefined}} \text{TYPEPRINT} \\
\\
\frac{\Gamma \vdash e_1 : \mathbf{string} \quad \Gamma \vdash e_2 : \mathbf{string}}{\Gamma \vdash e_1 + e_2 : \mathbf{string}} \text{TYPEPLUSSTR} \\
\\
\frac{\Gamma \vdash e_1 : \mathbf{number} \quad \Gamma \vdash e_2 : \mathbf{number} \quad bop \in \{+, *, /, -\}}{\Gamma \vdash e_1 \text{ } bop \text{ } e_2 : \mathbf{number}} \text{TYPEARITH} \\
\\
\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau \quad \tau \in \{\mathbf{number}, \mathbf{string}\} \quad bop \in \{>, >=, <, <=\}}{\Gamma \vdash e_1 \text{ } bop \text{ } e_2 : \mathbf{bool}} \text{TYPEINEQUAL} \\
\\
\frac{\Gamma \vdash e_d : \tau_d \quad \Gamma' = \Gamma[x \mapsto (\mathbf{mut}, \tau_d)] \quad \Gamma' \vdash e_b : \tau_b}{\Gamma \vdash \mathbf{mut } x = e_d ; e_b : \tau_b} \text{TYPEDECL} \\
\\
\frac{x \in \text{dom}(\Gamma) \quad \Gamma(x) = (\mathbf{mut}, \tau)}{\Gamma \vdash x : \tau} \text{TYPEVAR} \\
\\
\frac{\begin{array}{l} \Gamma' = \Gamma[x_1 \mapsto (\mathbf{const}, \tau_1)] \dots [x_n \mapsto (\mathbf{const}, \tau_n)] \\ \Gamma' \vdash e : \tau \quad \tau' = (\tau_1, \dots, \tau_n) \Rightarrow \tau \end{array}}{\Gamma \vdash \mathbf{function}(x_1 : \tau_1, \dots, x_n : \tau_n) e : \tau'} \text{TYPEFUN} \\
\\
\frac{\Gamma \vdash e_1 : \tau_1 \quad \dots \quad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash \{\mathbf{mut}_1 f_1 : e_1, \dots, \mathbf{mut}_n f_n : e_n\} : \{\mathbf{mut}_1 f_1 : \tau_1, \dots, \mathbf{mut}_n f_n : \tau_n\}} \text{TYPEOBJ} \\
\\
\frac{\begin{array}{l} \Gamma \vdash e : \{\mathbf{mut}_1 f_1 : \tau_1, \dots, \mathbf{mut}_n f_n : \tau_n\} \\ f = f_i \quad \tau = \tau_i \quad i \in [1, n] \end{array}}{\Gamma \vdash e.f : \tau} \text{TYPEDEREFELD}
\end{array}$$

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

$$\begin{array}{c}
\frac{\Gamma \vdash e : (\tau_1, \dots, \tau_n) \Rightarrow \tau \quad \text{for all } i: \quad \Gamma \vdash e_i : \tau'_i \text{ and } \tau'_i <: \tau_i}{\Gamma \vdash e(e_1, \dots, e_n) : \tau} \text{TYPECALL} \\
\\
\frac{\Gamma' = \Gamma[x_1 \mapsto (\mathbf{const}, \tau_1)] \dots [x_n \mapsto (\mathbf{const}, \tau_n)] \quad \tau = (\tau_1, \dots, \tau_n) \Rightarrow \tau_0 \quad \Gamma' \vdash e : \tau'_0 \quad \tau'_0 <: \tau_0}{\Gamma \vdash \mathbf{function} (x_1 : \tau_1, \dots, x_n : \tau_n) : \tau_0 \ e : \tau} \text{TYPEFUNANN} \\
\\
\frac{\Gamma' = \Gamma[x \mapsto \tau][x_1 \mapsto (\mathbf{const}, \tau_1)] \dots [x_n \mapsto (\mathbf{const}, \tau_n)] \quad \tau = (\tau_1, \dots, \tau_n) \Rightarrow \tau_0 \quad \Gamma' \vdash e : \tau'_0 \quad \tau'_0 <: \tau_0}{\Gamma \vdash \mathbf{function} x (x_1 : \tau_1, \dots, x_n : \tau_n) : \tau_0 \ e : \tau'} \text{TYPEFUNREC} \\
\\
\frac{\Gamma(x) = (\mathbf{var}, \tau') \quad \Gamma \vdash e : \tau \quad \tau <: \tau'}{\Gamma \vdash x = e : \tau} \text{TYPEASSIGNVAR} \\
\\
\frac{\Gamma \vdash e_1 : \{ \dots \mathbf{var} f : \tau' \dots \} \quad \Gamma \vdash e_2 : \tau \quad \tau <: \tau'}{\Gamma \vdash e_1.f = e_2 : \tau} \text{TYPEASSIGNFLD} \\
\\
\frac{\Gamma \vdash e_1 : \mathbf{bool} \quad \Gamma \vdash e_2 : \tau_2 \quad \Gamma \vdash e_3 : \tau_3 \quad \tau_2 \sqcup \tau_3 = \tau}{\Gamma \vdash e_1 ? e_2 : e_3 : \tau} \text{TYPEIF} \\
\\
\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_1 \sqcup \tau_2 \neq \mathbf{any} \quad \tau_1, \tau_2 \text{ have no function types} \quad bop \in \{==, !=\}}{\Gamma \vdash e_1 \text{ } bop \text{ } e_2 : \mathbf{bool}} \text{TYPEEQUAL}
\end{array}$$

Figure 4: Type checking rules of JAKARTAScript that involve subtyping

$$\begin{array}{c}
\overline{\tau <: \mathbf{any}} \text{SUBANY} \quad \overline{\tau <: \tau} \text{SUBREFL} \\
\\
\frac{\tau <: \tau' \quad \text{for all } i: \tau'_i <: \tau_i}{((\tau_1, \dots, \tau_n) \Rightarrow \tau) <: ((\tau'_1, \dots, \tau'_n) \Rightarrow \tau')} \text{SUBFUN} \\
\\
\frac{\{g_1, \dots, g_m\} \subseteq \{f_1, \dots, f_n\} \quad \text{for all } i, j, \text{ if } f_i = g_j, \text{ then } mut'_j = \mathbf{const} \text{ and } \tau_i <: \tau'_j \quad \text{or } mut_i = mut'_j \text{ and } \tau_i = \tau'_j}{\{mut_1 f_1 : \tau_1, \dots, mut_n f_n : \tau_n\} <: \{mut'_1 g_1 : \tau'_1, \dots, mut'_m g_m : \tau'_m\}} \text{SUBOBJ}
\end{array}$$

Figure 5: Subtyping rules of JAKARTAScript

$$\begin{array}{c}
\frac{\tau \in \{\mathbf{bool}, \mathbf{number}, \mathbf{string}, \mathbf{Undefined}\}}{\tau \sqcup \tau = \tau} \text{JOINBASIC=} \\
\\
\frac{\tau \neq \tau' \quad \tau \in \{\mathbf{bool}, \mathbf{number}, \mathbf{string}, \mathbf{Undefined}\}}{\tau \sqcup \tau' = \mathbf{any}} \text{JOINANY}_1 \\
\\
\frac{\tau \neq \tau' \quad \tau' \in \{\mathbf{bool}, \mathbf{number}, \mathbf{string}, \mathbf{Undefined}\}}{\tau \sqcup \tau' = \mathbf{any}} \text{JOINANY}_2 \\
\\
\frac{}{\{\} \sqcup \{\overline{mut_g g : \tau_g}\} = \{\}} \text{JOINOBJEMP} \\
\\
\frac{(\tau_h \neq \tau'_h \text{ or } \overline{mut_h} \neq \overline{mut'_h}) \quad \{\overline{mut_f f : \tau_f}\} \sqcup \{\overline{mut_g g : \tau_g}, \overline{mut_{g'} g' : \tau_{g'}}\} = \{\overline{mut_k k : \tau_k}\} \quad \tau_h \sqcup \tau'_h = \tau''_h \quad \tau'' = \{\mathbf{const } h : \tau''_h, \overline{mut_k k : \tau_k}\}}{\{\overline{mut_h h : \tau_h}, \overline{mut_f f : \tau_f}\} \sqcup \{\overline{mut_g g : \tau_g}, \overline{mut'_h h : \tau'_h}, \overline{mut_{g'} g' : \tau_{g'}}\} = \tau''} \text{JOINOBJMUT}_{\neq} \\
\\
\frac{\{\overline{mut_f f : \tau_f}\} \sqcup \{\overline{mut_g g : \tau_g}, \overline{mut_{g'} g' : \tau_{g'}}\} = \{\overline{mut_k k : \tau_k}\} \quad \tau'' = \{\mathbf{var } h : \tau_h, \overline{mut_k k : \tau_k}\}}{\{\mathbf{var } h : \tau_h, \overline{mut_f f : \tau_f}\} \sqcup \{\overline{mut_g g : \tau_g}, \mathbf{var } h : \tau_h, \overline{mut_{g'} g' : \tau_{g'}}\} = \tau''} \text{JOINOBJVAR}= \\
\\
\frac{h \notin \{\bar{g}\} \quad \{\overline{mut_f f : \tau_f}\} \sqcup \{\overline{mut_g g : \tau_g}\} = \tau''}{\{\overline{mut_h h : \tau_h}, \overline{mut_f f : \tau_f}\} \sqcup \{\overline{mut_g g : \tau_g}\} = \tau''} \text{JOINOBJNO} \\
\\
\frac{\text{for all } i \in [1, n]: \tau_i \sqcap \tau'_i = \tau''_i \quad \tau_0 \sqcup \tau'_0 = \tau''_0 \quad \tau'' = (\tau''_1, \dots, \tau''_n) \Rightarrow \tau''_0}{((\tau_1, \dots, \tau_n) \Rightarrow \tau_0) \sqcup ((\tau'_1, \dots, \tau'_n) \Rightarrow \tau'_0) = \tau''} \text{JOINFUNMEET} \\
\\
\frac{n \neq m}{((\tau_1, \dots, \tau_n) \Rightarrow \tau_0) \sqcup ((\tau'_1, \dots, \tau'_m) \Rightarrow \tau'_0) = \mathbf{any}} \text{JOINFUNANY}_{\neq} \\
\\
\frac{\tau_i \sqcap \tau'_i = \perp \quad i \in [1, n]}{((\tau_1, \dots, \tau_n) \Rightarrow \tau_0) \sqcup ((\tau'_1, \dots, \tau'_n) \Rightarrow \tau'_0) = \mathbf{any}} \text{JOINFUNANY}= \\
\\
\frac{}{\{\overline{mut_f f : \tau_f}\} \sqcup ((\tau_1) \Rightarrow \tau_2) = \mathbf{any}} \text{JOINOBJFUN} \\
\\
\frac{}{((\tau_1) \Rightarrow \tau_2) \sqcup \{\overline{mut_f f : \tau_f}\} = \mathbf{any}} \text{JOINFUNOBJ}
\end{array}$$

Figure 6: Rules for computing joins

$$\begin{array}{c}
\frac{\tau \in \{\mathbf{bool}, \mathbf{number}, \mathbf{string}, \mathbf{Undefined}\}}{\tau \sqcap \tau = \tau} \text{MEETBASIC=} \\
\\
\frac{}{\mathbf{any} \sqcap \tau = \tau} \text{MEETANY}_1 \quad \frac{}{\tau \sqcap \mathbf{any} = \tau} \text{MEETANY}_2 \\
\\
\frac{}{\{\} \sqcap \{\overline{mut_g g : \tau_g} = \overline{mut_g g : \tau_g}\}} \text{MEETOBJEMP} \\
\\
\frac{\begin{array}{c} \overline{mut_f f : \tau_f} \sqcap \{\overline{mut_g g : \tau_g}, \overline{mut_{g'} g' : \tau_{g'}}\} = \{\overline{mut_k k : \tau_k}\} \\ \tau_h \sqcap \tau'_h = \tau''_h \quad \tau = \{\mathbf{const} h : \tau''_h, \overline{mut_k k : \tau_k}\} \end{array}}{\{\mathbf{const} h : \tau_h, \overline{mut_f f : \tau_f}\} \sqcap \{\overline{mut_g g : \tau_g}, \mathbf{const} h : \tau'_h, \overline{mut_{g'} g' : \tau_{g'}}\} = \tau} \text{MEETOBJCONST} \\
\\
\frac{\begin{array}{c} \overline{mut_f f : \tau_f} \sqcap \{\overline{mut_g g : \tau_g}, \overline{mut_{g'} g' : \tau_{g'}}\} = \{\overline{mut_k k : \tau_k}\} \\ \tau = \{\mathbf{var} h : \tau_h, \overline{mut_k k : \tau_k}\} \end{array}}{\{\mathbf{var} h : \tau_h, \overline{mut_f f : \tau_f}\} \sqcap \{\overline{mut_g g : \tau_g}, \mathbf{var} h : \tau_h, \overline{mut_{g'} g' : \tau_{g'}}\} = \tau} \text{MEETOBJVAR=} \\
\\
\frac{\begin{array}{c} \overline{mut_f f : \tau_f} \sqcap \{\overline{mut_g g : \tau_g}, \overline{mut_{g'} g' : \tau_{g'}}\} = \{\overline{mut_k k : \tau_k}\} \\ \tau_h \neq \tau'_h \quad \overline{mut_h h : \tau_h} \neq \overline{mut'_h h : \tau'_h} \quad \tau_h \sqcap \tau'_h = \tau''_h \quad \tau = \{\mathbf{var} h : \tau''_h, \overline{mut_k k : \tau_k}\} \end{array}}{\{\overline{mut_h h : \tau_h}, \overline{mut_f f : \tau_f}\} \sqcap \{\overline{mut_g g : \tau_g}, \overline{mut'_h h : \tau'_h}, \overline{mut_{g'} g' : \tau_{g'}}\} = \tau} \text{MEETOBJMUT}\neq \\
\\
\frac{\begin{array}{c} \overline{mut_f f : \tau_f} \sqcap \{\overline{mut_g g : \tau_g}\} = \{\overline{mut_k k : \tau_k}\} \\ h \notin \{\bar{g}\} \quad \tau'' = \{\overline{mut_h h : \tau_h}, \overline{mut_k k : \tau_k}\} \end{array}}{\{\overline{mut_h h : \tau_h}, \overline{mut_f f : \tau_f}\} \sqcap \{\overline{mut_g g : \tau_g}\} = \tau''} \text{MEETOBJNO} \\
\\
\frac{\begin{array}{c} \text{for all } i \in [1, n]: \tau_i \sqcup \tau'_i = \tau''_i \\ \tau_0 \sqcap \tau'_0 = \tau''_0 \quad \tau'' = (\tau''_1, \dots, \tau''_n) \Rightarrow \tau''_0 \end{array}}{(\tau_1, \dots, \tau_n) \Rightarrow \tau_0 \sqcap (\tau'_1, \dots, \tau'_n) \Rightarrow \tau'_0 = \tau''} \text{MEETFUNJOIN}
\end{array}$$

Figure 7: Rules for computing meets