Database Languages and their Compilers

Prof. Dr. Torsten Grust

Database Systems Research Group
U Tübingen

Winter 2010

3 Variable Scopes and Type Inference for SQL

- We have gone a long way to recognize syntactically correct SQL queries but there is still more work to do to further analyze the meaning of queries.

- This chapter will discuss how we can get control of two **semantic properties** of SQL which go beyond syntax:
  - SQL **variable scopes**, and
  - the **typing** of SQL (sub)queries.

- MCC will turn out to be an intermediate query representation that allows us to treat both issues relatively easy.

3.1 Variables in SQL

- From the viewpoint of the query processor, variables in SQL are a language feature of core importance.

- (Im)proper usage of variables in SQL *can* make a difference. Consider:

$$\begin{array}{rl} \texttt{SELECT} & \texttt{COUNT}(*) \\ \texttt{FROM} & R\ x \\ \texttt{WHERE} & \texttt{FORALL}\ y\ \texttt{IN}\ S : x.A > y.A \end{array}$$

  - If we replace tuple variable y by x (e.g., through a user typo), this query computes something completely different.

> ✎ **Static Query Analysis**
> Trading y for x in the above query has consequences that we can account for **statically**, i.e., before query execution begins.

- Full analysis of queries with variables seems to be tricky enough; on top of that, SQL allows users to be lax when it comes to variable usage.

- Consider two relations $X :: \mathit{bag}\ (x : \mathbb{N}, y : \mathbb{B})$ and $Y :: \mathit{bag}\ (y : \mathbb{S}, z : \mathbb{S})$.

  - **Shadowing I**:

$$\begin{array}{rl} \texttt{SELECT} & x, y \\ \texttt{FROM} & X\ x, \big(\texttt{SELECT}\ y\ \texttt{FROM}\ Y\big) \end{array}$$

    The attribute **references** y **are ambiguous**: is this y of X (i.e., $x.y$) or y of Y? (SQL: y of relation Y.)

  - **Shadowing II**:

$$\begin{array}{rl} \texttt{SELECT} & x, y \\ \texttt{FROM} & X\ x, \big(\texttt{SELECT}\ z\ \texttt{FROM}\ Y\big) \end{array}$$

    Usage of tuple variable x **collides** with presence of attribute x of the same name. (SQL: x refers to the attribute of X, not the tuple variable.)

3.1.1 Variable Environments

- The problem is tractable, though. The minimalistic syntactic structure of MCC and its equally simple **variable scoping rules** come to the rescue.
  - We make use of a simple data structure, the **variable environment** E, which we associate with each (sub)expression in MCC:

$$E \quad = \quad \{v_1 \mapsto \tau_1, \ldots, v_n \mapsto \tau_n\} \ .$$

  - An entry $v_i \mapsto \tau_i$ in the environment specifies that a variable $v_i :: \tau_i$ **is in scope**.
  - If a new variable $v :: \tau$ comes into scope, we **add** $\{v \mapsto \tau\}$ to E:

$$E + \{v \mapsto \tau\} \quad = \quad \{v_1 \mapsto \tau_1, \ldots, v_n \mapsto \tau_n, v \mapsto \tau\}$$

  - A **lookup** for v in $\{\ldots, v \mapsto \tau, \ldots, v \mapsto \tau', \ldots\}$ returns τ'.
    Thus, $E + E' \neq E' + E$.

- Comprehensions interact with the environment they are evaluated in in a simple manner. Let E be the current environment in which we evaluate

$$[\, e \mid q_1, \ldots, q_n \,]^M \quad .$$

Then ("*the environment is augmented from left to right*"):

- q_1 is evaluated in E;
- if the current environment is E and $q_i = v_i \leftarrow e_i$, with $e_i :: bag\ \tau_i$, then $q_{i+1}, \ldots, q_n$ are evaluated in environment $E + \{v_i \mapsto \tau_i\}$;
- e is evaluated in the environment associated with q_n.

> ✎ **Which value is computed by the monoid comprehension below?**
>
> $$[\, x \mid x \leftarrow \{1, 2, 3\}, x \leq 2, x \leftarrow \{true, false\} \,]^{bag}$$

- To return to our example of variable shadowing (I), here is how we can make use of environments to resolve the ambiguity:
  - Original SQL query:

$$\begin{array}{ll}\texttt{SELECT} & x, y \\ \quad \texttt{FROM} & X\ x, \big(\texttt{SELECT}\ y\ \texttt{FROM}\ Y\big)\end{array}$$

> ✎ **Resolve variable reference ambiguities by use of environments:**
> Equivalent MCC expression:
>
> $$[\,(x : x\ ,\ y : y) \mid x \leftarrow X\ ,\ R_2 \leftarrow [\,(y\ :\ y) \mid Y_1 \leftarrow Y\,]^{bag}\,]^{bag}$$
>
> ③ ① ②
>
> (Check the environment at points ⓘ, $i = 1 \ldots 3$.)

3.2 The Type of an SQL Query

- **Type checking** concerns query processing in much the same way as the compilation of programming languages.
  - Try to **detect "semantic nonsense"**, e.g.:

```
SELECT  x.A
  FROM  (VALUES 1, TRUE, "foo") x
 WHERE  EXISTS z IN x : x + z ;
```

  - types help to **prepare query execution** (allocation of buffers for (intermediate) results of correct size and shape);
  - static type checking helps to **avoid runtime sanity checks**.

- Once more we can employ MCC as a concise intermediate representation for queries.
  - Instead of type checking SQL directly we type check the simpler MCC. The basic observation is that we have, for any SQL query q:

$$q :: \tau \quad \Leftrightarrow \quad \mathbb{T}(q) :: \tau \ .$$

3.2.1 Type Inference

- **Inferring the type** of an MCC expression is actually a simple undertaking given that we already know the types of its subexpressions. Consider:
  - Given $e_1 :: \mathbb{N}$ and $e_2 :: \mathbb{N}$, we have $(e_1 + e_2) :: \mathbb{N}$;
  - Given $e :: \tau$, we have $[\, e \mid \ldots]^{set} :: set\ \tau$;
  - Given $e :: bag\ \tau$ and $[\ldots \mid \ldots, v \leftarrow e, p, \ldots]^{all}$, p is evaluated in an environment of the form $E + \{v \mapsto \tau\}$;
  - Given an environment of the form $E + \{v \mapsto \tau\}$, we have $v :: \tau$.

- This type inference process is rooted in trivial type assignments such as

$$42 :: \mathbb{N} \quad \text{or} \quad \texttt{"foo"} :: \mathbb{S} \; .$$

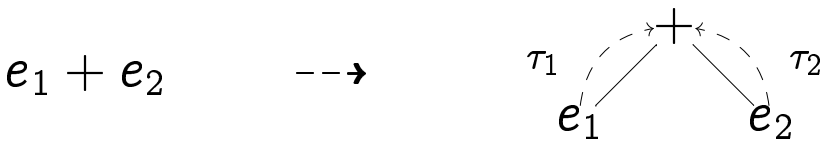
- If we want to implement type inference using such a scheme, clearly we have to proceed **bottom-up**.

- The ultimate goal is to annotate any subexpression (including the overall expression) with its type τ of the following shape:

$$\begin{array}{rcll} \tau & \rightarrow & \mathbb{N} & \\ & | & \mathbb{S} & \\ & | & \mathbb{B} & \\ & | & \mathit{bag}\ \tau & \\ & | & \mathit{set}\ \tau & \\ & | & \mathit{list}\ \tau & \\ & | & (l_1 : \tau, \ldots, l_n : \tau) & (l_i \text{ attribute identifier}) \end{array}$$

3.3 Semantic Attributes

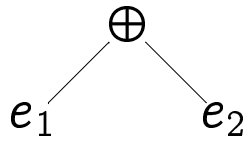
- Leaving the variable scoping business aside for a minute, for type inference purposes, it will be sensible to denote an MCC expression by an equivalent **expression tree**, e.g.:



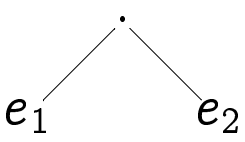
  - Inferring types bottom up then simply means to **propagate type information from the leaves up to the root** of the expression tree (i.e., along the dashed arrows).

- Luckily, MCC is so simple that we can define the structure of expression trees on a single slide.

$e_1 \oplus e_2$

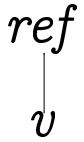


$e1.e2$

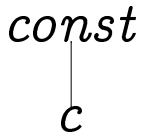


$\neg\ e$

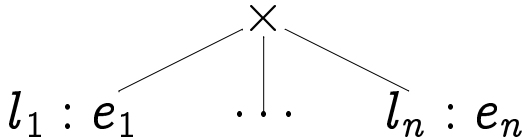
v (variable)



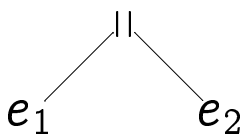
c (constant)



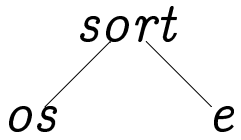
$(l_1 : e_1, \ldots, l_n : e_n)$



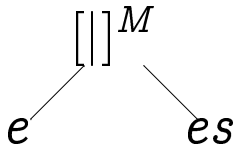
$e_1 \mid\mid e_2$



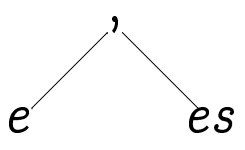
$sort\ os\ e$



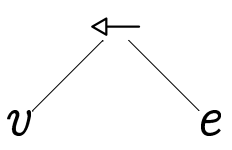
$\left[\, e \mid es \,\right]^M$



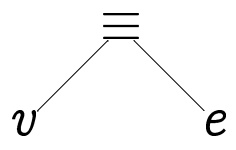
e, es



$v \leftarrow e$



$v \equiv e$



✎ **What would be the shape of the expression tree for the comprehension below?**

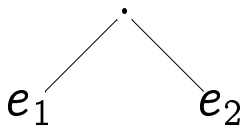
$$[\,(x : x, y : y) \mid x \leftarrow X, R_2 \leftarrow [\,(y : y) \mid y \leftarrow Y\,]^{bag}\,]^{bag}$$

- We can now organize our type inference process by simply **attaching an attribute named** *type* **to each node** of the expression tree.
  - In accordance to our plan to proceed **bottom up**, we can refer to the *type* attribute of child nodes when we try to determine the *type* for a current node (*type* is said to be **synthesized**), e.g.:


```
¬.type ←
  IF e.type = 𝔹 THEN
   | 𝔹
  ELSE
   | type-↯
```

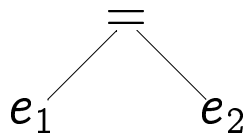
  - If the flow of control reaches *type*-↯ we have detected **a type error** and abort the type inference process.

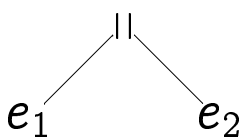
■ Or, quite similarly:

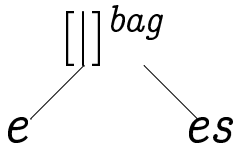


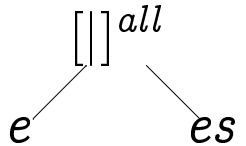
$$
\begin{array}{l}
\cdot.type \leftarrow \\
\quad \texttt{IF } e_1.type = (l_1 : \tau_1, \ldots, l_n : \tau_n) \texttt{ THEN} \\
\quad\quad \texttt{IF } e_2 = l_i \texttt{ THEN} \\
\quad\quad\quad \mid\ \tau_i \\
\quad\quad \texttt{ELSE} \\
\quad\quad\quad \texttt{IF } e_2 = * \texttt{ THEN} \\
\quad\quad\quad\quad \mid\ e_1.type \\
\quad\quad\quad \texttt{ELSE} \\
\quad\quad\quad\quad \lfloor\ type\text{-}\lightning \\
\quad \texttt{ELSE} \\
\quad\quad \lfloor\ type\text{-}\lightning
\end{array}
$$

✎ **Devise the type inference rules for the MCC expression trees below:**



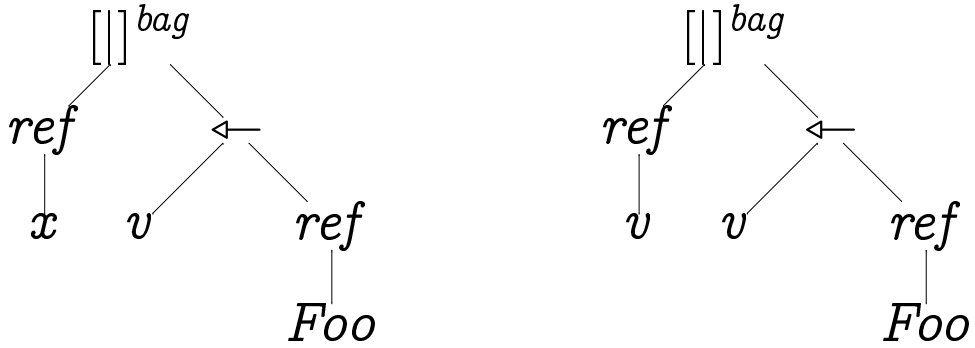






- We need to be more careful as soon as we come across *ref* nodes.
  - Consider the two comprehensions and their respective expression trees below(let $Foo :: bag\ (x : \mathbb{N})$ denote a relation residing in the DBMS):

$$[\, x \mid v \leftarrow Foo \,]^{bag} \qquad [\, v \mid v \leftarrow Foo \,]^{bag}$$

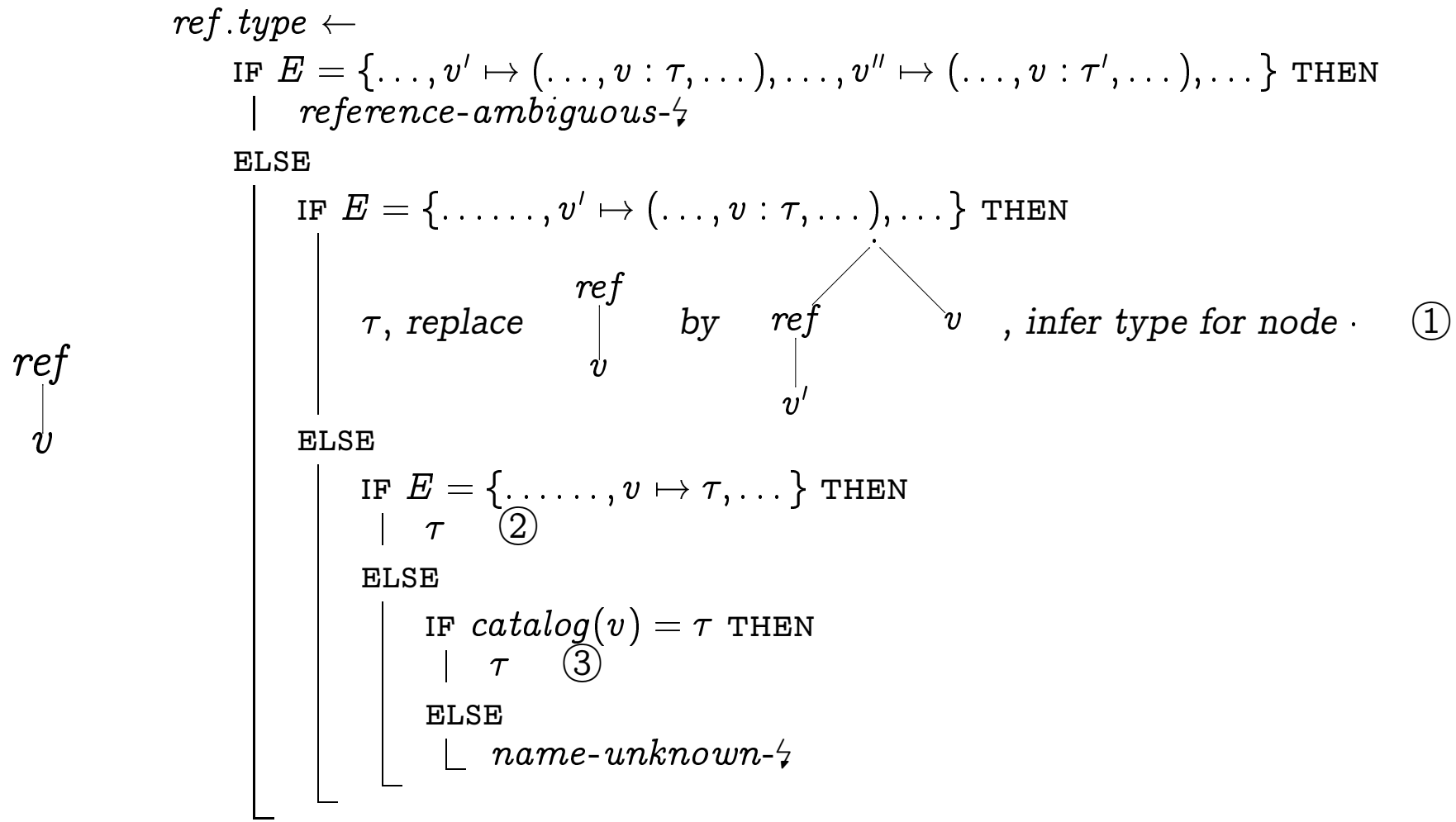


① $\overset{ref}{\underset{x}{|}}$: reference to a **column name** of a variable in scope;

② $\overset{ref}{\underset{v}{|}}$: reference to a **variable** in scope;

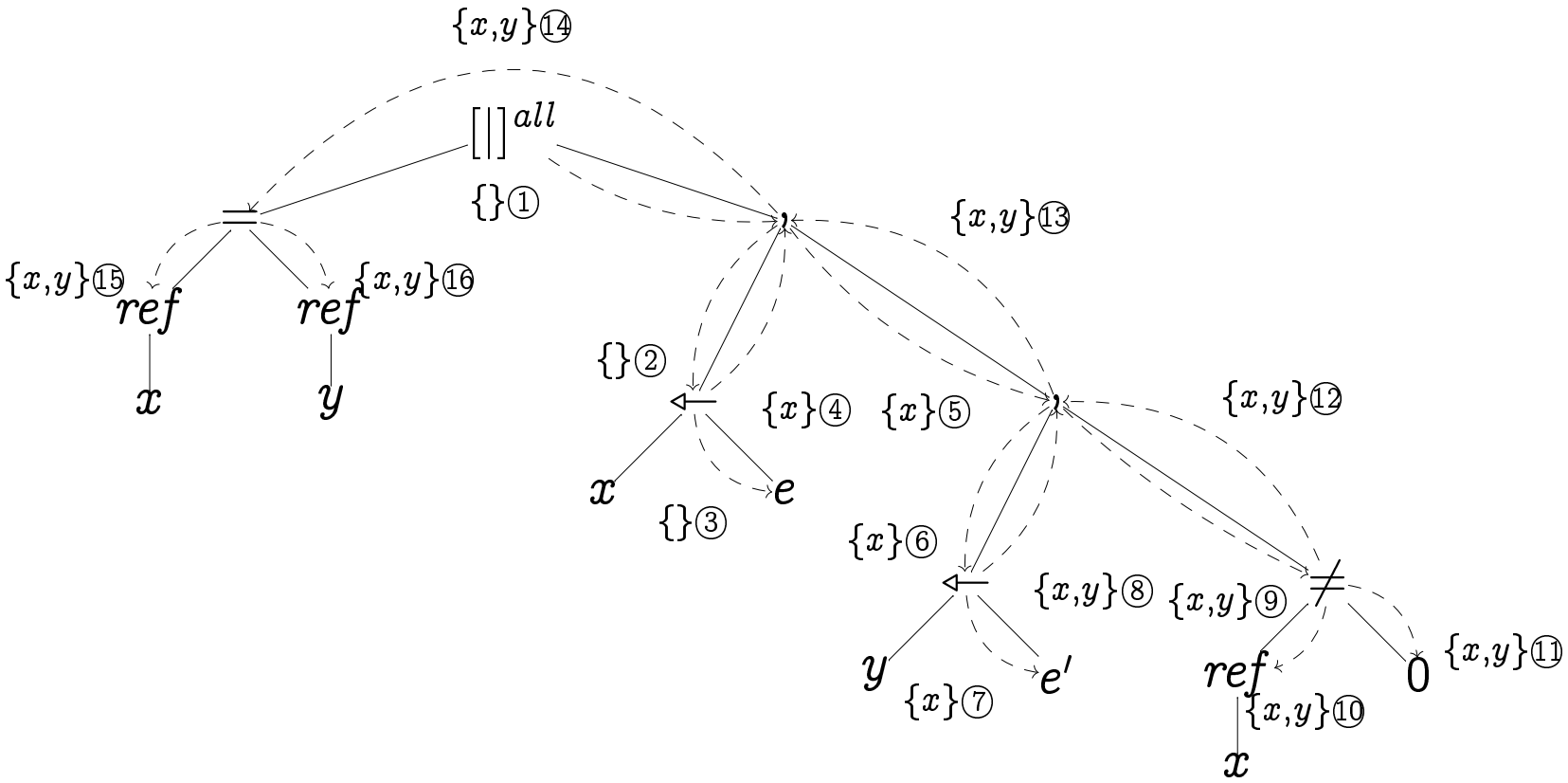
③ $\overset{ref}{\underset{Foo}{|}}$: reference to a **relation registered in the DBMS catalog**.

- At this stage in query analysis we are not yet able to distinguish the three cases.
  - What we need is access to the current **environment** E during type inference:



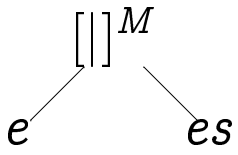
- To make the current variable environment accessible let us attach another **semantic attribute**, *env*, to each node in the MCC expression trees.
  - Here is an example of how the value of *env* has to be propagated:

$$\left[\, x = y \mid x \leftarrow e, y \leftarrow e', x \neq 0 \,\right]^{all}$$

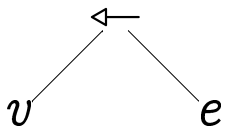


- Note that—for almost all node types—the *env* attribute is propagated **top-down** during this process. Attribute *env* is said to be **inherited**.

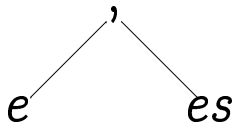
- An exception to this are the node types $[|]^M$, $\Leftarrow$, and $(,)$ which represent the comprehension construct in MCC:



es.env $\leftarrow$ $[|]^M$*.env* ;
do environment propagation in es (may update es.env) ;
e.env $\leftarrow$ *es.env* ;



e.env $\leftarrow$ $\Leftarrow$ *.env* ;
do environment propagation in e ;
v.type $\leftarrow$ IF *e.type* $=$ *bag* τ THEN τ ELSE *type-*$\lightning$;
$\Leftarrow$ *.env* $\leftarrow$ $\Leftarrow$ *.env* $+ \{v \mapsto v.type\}$;



e.env ← *, .env* ;
do environment propagation in e (may update e.env) ;
es.env ← *e.env* ;
do environment propagation in es (may update es.env) ;
, .env ← *es.env* ;

- We could have done without explicit instructions like *do environment propagation in ...*, if we adopt a simple rule:

  > The occurrence of a semantic attribute (of a child node) on the right-hand side of an assignment ($\leftarrow$) implies that his node must have been visited prior to the assignment.

- Traversing expression (or parse) trees to annotate nodes with semantic attributes is a powerful and well-understood technique in Computer Science: see **attribute grammars**.

Contents