Type Systems

- A *type system* defines a set of types and rules to assign types to programming language constructs
- Informal type system rules, for example "*if both operands of addition are of type integer, then the result is of type integer*"
- Formal type system rules: Post systems

Type Rules in Post System Notation

$$\frac{\rho(v) = \tau}{\rho \vdash\ v : \tau}$$

Type judgments
$e : \tau$
where e is an expression and τ is a type

$$\frac{\rho(v) = \tau \qquad \rho \vdash e : \tau}{\rho \vdash\ v \mathbin{\mathbf{:=}} e : \mathit{void}}$$

Environment ρ maps objects v to types τ:
$\rho(v) = \tau$

$$\frac{\rho \vdash e_1 : \mathit{integer} \qquad \rho \vdash e_2 : \mathit{integer}}{\rho \vdash\ e_1 \mathbin{\mathbf{+}} e_2 : \mathit{integer}}$$

Type System Example

Environment ρ is a set of $\langle$*name*, *type*$\rangle$ pairs, for example:

$$\rho = \{\ \langle \mathbf{x},\mathit{integer}\rangle, \langle \mathbf{y},\mathit{integer}\rangle, \langle \mathbf{z},\mathit{char}\rangle, \langle 1,\mathit{integer}\rangle, \langle 2,\mathit{integer}\rangle\ \}$$

From ρ and rules we can check the validity of typed expressions:

type checking = theorem proving

The proof that **x := y + 2** is typed correctly:

$$\frac{\rho(\mathbf{x}) = \mathit{integer} \qquad \dfrac{\dfrac{\rho(\mathbf{y}) = \mathit{integer}}{\rho \vdash \mathbf{y} : \mathit{integer}} \qquad \dfrac{\rho(\mathbf{2}) = \mathit{integer}}{\rho \vdash \mathbf{2} : \mathit{integer}}}{\rho \vdash \mathbf{y + 2} : \mathit{integer}}}{\rho \vdash \mathbf{x := y + 2} : \mathit{void}}$$

A Simple Language Example

$$
\begin{array}{lll}
P & \rightarrow & D \;\mathbf{;}\; S \\
D & \rightarrow & D \;\mathbf{;}\; D \\
 & | & \mathbf{id} \;\mathbf{:}\; T \\
T & \rightarrow & \mathbf{boolean} \\
 & | & \mathbf{char} \\
 & | & \mathbf{integer} \\
 & | & \mathbf{array} \;\mathbf{[}\; \mathbf{num} \;\mathbf{]}\; \mathbf{of} \; T \\
 & | & \mathbf{\wedge} \; T \\
S & \rightarrow & \mathbf{id} \;\mathbf{:=}\; E \\
 & | & \mathbf{if} \; E \; \mathbf{then} \; S \\
 & | & \mathbf{while} \; E \; \mathbf{do} \; S \\
 & | & S \;\mathbf{;}\; S
\end{array}
$$

$$
\begin{array}{lll}
E & \rightarrow & \mathbf{true} \\
 & | & \mathbf{false} \\
 & | & \mathbf{literal} \\
 & | & \mathbf{num} \\
 & | & \mathbf{id} \\
 & | & E \; \mathbf{and} \; E \\
 & | & E \;\mathbf{+}\; E \\
 & | & E \;\mathbf{[}\; E \;\mathbf{]} \\
 & | & E \; \mathbf{\wedge}
\end{array}
$$

Pointer to T

Pascal-like pointer dereference operator

Simple Language Example: Declarations

$D \rightarrow$ **id :** T { $addtype($**id**$.\text{entry}, T.\text{type})$ }
$T \rightarrow$ **boolean** { $T.\text{type} := boolean$ }
$T \rightarrow$ **char** { $T.\text{type} := char$ }
$T \rightarrow$ **integer** { $T.\text{type} := integer$ }
$T \rightarrow$ **array [num] of** T_1 { $T.\text{type} := array(1..$**num**$.\text{val}, T_1.\text{type})$ }
$T \rightarrow$ **^** T_1 { $T.\text{type} := pointer(T_1)$

Parametric types:
type constructor

Simple Language Example: Checking Statements

$$\frac{\rho(v) = \tau \qquad \rho \vdash e : \tau}{\rho \vdash v \mathbin{\mathbf{:=}} e : \mathit{void}}$$

$S \rightarrow$ **id :=** E { S.type := (**if id**.type = E.type **then** *void* **else** *type_error*) }

Note: the type of **id** is determined by scope's environment:
id.type = *lookup*(**id**.entry)

Simple Language Example: Checking Statements (cont'd)

$$\frac{\rho \vdash e : \mathit{boolean} \qquad \rho \vdash s : \tau}{\rho \vdash \textbf{if } e \textbf{ then } s : \tau}$$

$S \rightarrow$ **if** E **then** S_1 { S.type := (**if** E.type = *boolean* **then** S_1.type **else** *type_error*) }

Simple Language Example: Statements (cont'd)

$$\frac{\rho \vdash e : \mathit{boolean} \qquad \rho \vdash s : \tau}{\rho \vdash \textbf{while}\ e\ \textbf{do}\ s : \tau}$$

$S \rightarrow$ **while** E **do** S_1 { S.type := (**if** E.type = *boolean* **then** S_1.type **else** *type_error*) }

Simple Language Example: Checking Statements (cont'd)

$$\frac{\rho \vdash s_1 : \mathit{void} \qquad \rho \vdash s_2 : \mathit{void}}{\rho \vdash s_1 \mathbin{\mathbf{;}} s_2 : \mathit{void}}$$

$S \rightarrow S_1$ **;** S_2 { S.type := (**if** S_1.type = *void* **and** S_2.type = *void* **then** *void* **else** *type_error*) }

Simple Language Example: Checking Expressions

$$\frac{\rho(v) = \tau}{\rho \vdash \ v : \tau}$$

$E \rightarrow$ **true**	{ E.type = *boolean* }
$E \rightarrow$ **false**	{ E.type = *boolean* }
$E \rightarrow$ **literal**	{ E.type = *char* }
$E \rightarrow$ **num**	{ E.type = *integer* }
$E \rightarrow$ **id**	{ E.type = *lookup*(**id**.entry) }
…	

Simple Language Example: Checking Expressions (cont'd)

$$\frac{\rho \vdash e_1 : \mathit{integer} \qquad \rho \vdash e_2 : \mathit{integer}}{\rho \vdash e_1 \mathbf{+} e_2 : \mathit{integer}}$$

$E \rightarrow E_1 \mathbf{+} E_2$ { E.type := (**if** E_1.type = *integer* **and** E_2.type = *integer*
then *integer* **else** *type_error*) }

Simple Language Example: Checking Expressions (cont'd)

$$\frac{\rho \vdash e_1 : \mathit{boolean} \qquad \rho \vdash e_2 : \mathit{boolean}}{\rho \vdash e_1 \textbf{ and } e_2 : \mathit{boolean}}$$

$E \rightarrow E_1$ **and** E_2 { E.type := (**if** E_1.type = *boolean* **and** E_2.type = *boolean*
then *boolean* **else** *type_error*) }

Simple Language Example: Checking Expressions (cont'd)

$$\frac{\rho \vdash e_1 : \mathit{array}(s, \tau) \quad \rho \vdash e_2 : \mathit{integer}}{\rho \vdash e_1[e_2] : \tau}$$

$E \rightarrow E_1$ **[** E_2 **]** { E.type := (**if** E_1.type = $\mathit{array}(s, t)$ **and** E_2.type = $\mathit{integer}$ **then** t **else** $\mathit{type\_error}$) }

Note: parameter t is set with the unification of E_1.type = $\mathit{array}(s, t)$

Simple Language Example: Checking Expressions (cont'd)

$$\frac{\rho \vdash e : \mathit{pointer}(\tau)}{\rho \vdash e\,\hat{}\, : \tau}$$

$E \rightarrow E_1$ **^** $\{$ E.type := (**if** E_1.type = *pointer*(t) **then** t
else *type_error*) $\}$

Note: parameter t is set with the unification of
E_1.type = *pointer*(t)

A Simple Language Example: Functions

$T \rightarrow T$ **->** T

Function type declaration

$E \rightarrow E$ **(** E **)**

Function call

Example:

```
v : integer;
odd : integer -> boolean;
if odd(3) then
    v := 1;
```

Simple Language Example: Function Declarations

$$T \rightarrow T_1 \texttt{ -> } T_2 \ \{ \ T.\text{type} := \mathit{function}(T_1.\text{type}, T_2.\text{type}) \ \}$$

Parametric type:
type constructor

Simple Language Example: Checking Function Invocations

$$\frac{\rho \vdash e_1 : \mathit{function}(\sigma, \tau) \qquad \rho \vdash e_2 : \sigma}{\rho \vdash e_1(e_2) : \tau}$$

$E \rightarrow E_1$ **(** E_2 **)** { E.type := (**if** E_1.type = *function*(s, t) **and** E_2.type = s **then** t **else** *type_error*) }

Type Conversion and Coercion

- *Type conversion* is explicit, for example using type casts
- *Type coercion* is implicitly performed by the compiler to generate code that converts types of values at runtime (typically to *narrow* or *widen* a type)
- Both require a *type system* to check and infer types from (sub)expressions

Syntax-Directed Definitions for Type Checking in Yacc

```
%{
enum Types {Tint, Tfloat, Tpointer, Tarray, … };
typedef struct Type
{ enum Types type;
  struct Type *child; // at most one type parameter
} Type;
%}

%union
{ Type *typ;
}

%type <typ> expr

%%
…
```

Syntax-Directed Definitions for Type Checking in Yacc (cont'd)

```
…
%%

expr : expr '+' expr { if ($1->type != Tint
                         || $3->type != Tint)
                          semerror("non-int operands in +");
                        $$ = mkint();
                        emit(iadd);
                      }
```

Syntax-Directed Definitions for Type Coercion in Yacc

```
…
%%
expr : expr '+' expr
     { if ($1->type == Tint && $3->type == Tint)
         { $$ = mkint(); emit(iadd);
         }
       else if ($1->type == Tfloat && $3->type == Tfloat)
         { $$ = mkfloat(); emit(fadd);
         }
       else if ($1->type == Tfloat && $3->type == Tint)
         { $$ = mkfloat(); emit(i2f); emit(fadd);
         }
       else if ($1->type == Tint && $3->type == Tfloat)
         { $$ = mkfloat(); emit(swap); emit(i2f); emit(fadd);
         }
       else semerror("type error in +");
         $$ = mkint();
     }
```

Checking L-Values and R-Values in Yacc

```
%{
typedef struct Node
{ Type *typ;  // type structure
  int islval; // 1 if L-value
} Node;
%}

%union
{ Node *rec;
}

%type <rec> expr

%%
…
```

Checking L-Values and R-Values in Yacc

```
expr : expr '+' expr
         { if ($1->typ->type != Tint || $3->typ->type != Tint)
             semerror("non-int operands in +");
           $$->typ = mkint();
           $$->islval = FALSE;
           emit(...);
         }
     | expr '=' expr
         { if (!$1->islval || $1->typ != $3->typ)
             semerror("invalid assignment");
           $$->typ = $1->typ;
           $$->islval = FALSE;
           emit(...);
         }
     | ID
         { $$->typ = lookup($1);
           $$->islval = TRUE;
           emit(...);
         }
```

Type Inference and Polymorphic Functions

Many functional languages support polymorphic type systems

For example, the list length function in ML:

fun *length*(*x*) = **if** *null*(*x*) **then** 0 **else** *length*(*tl*(*x*)) + 1

length([“sun”, “mon”, “tue”]) + *length*([10,9,8,7])
returns 7

Type Inference and Polymorphic Functions

The type of **fun** *length* is:

$\forall \alpha .\text{list}(\alpha) \rightarrow \text{integer}$

We can infer the type of *length* from its body:

fun *length*(x) = **if** *null*(x) **then** 0 **else** *length*(*tl*(x)) + 1

where

null : $\forall \alpha .\text{list}(\alpha) \rightarrow \text{bool}$

tl : $\forall \alpha .\text{list}(\alpha) \rightarrow \text{list}(\alpha)$

and the return value is 0 or *length*(*tl*(x)) + 1, thus

length: $\forall \alpha .\text{list}(\alpha) \rightarrow \text{integer}$

Type Inference and Polymorphic Functions

Types of functions f are denoted by $\alpha \rightarrow \beta$ and the post-system rule to infer the type of $f(x)$ is:

$$\frac{\rho \vdash e_1 : \alpha \rightarrow \beta \qquad\qquad \rho \vdash e_2 : \alpha}{\rho \vdash e_1(e_2) : \beta}$$

The type of *length*([“a”, “b”]) is inferred by

$$\frac{\rho \vdash \mathit{length} : \forall \alpha .\text{list}(\alpha) \rightarrow \text{integer} \qquad \dfrac{\dots}{\rho \vdash [\text{“a”}, \text{“b”}] : \text{list(string)}}}{\rho \vdash \mathit{length}([\text{“a”}, \text{“b”}]) : \text{integer}}$$

Example Type Inference

Append concatenates two lists recursively:

fun *append*(x, y) = **if** *null*(x) **then** y
else *cons*(*hd*(x), *append*(*tl*(x), y))

where

null : $\forall \alpha .\mathrm{list}(\alpha) \to \mathrm{bool}$

hd : $\forall \alpha .\mathrm{list}(\alpha) \to \alpha$

tl : $\forall \alpha .\mathrm{list}(\alpha) \to \mathrm{list}(\alpha)$

cons : $\forall \alpha .(\alpha \times \mathrm{list}(\alpha)) \to \mathrm{list}(\alpha)$

Example Type Inference

fun *append*(*x*, *y*) = **if** *null*(*x*) **then** *y*

else *cons*(*hd*(*x*), *append*(*tl*(*x*), *y*))

The type of *append* : $\forall \sigma, \tau, \varphi.\ (\sigma \times \tau) \rightarrow \varphi$ is:

type of x : $\sigma = \text{list}(\alpha_1)$ from *null*(*x*)

type of y : $\tau = \varphi$ from *append*' s return type

return type of *append* : $\text{list}(\alpha_2)$ from return type of *cons*

and $\alpha_1 = \alpha_2$ because

$$\dfrac{\dfrac{\rho \vdash x : \text{list}(\alpha_1)}{\rho \vdash hd(x) : \alpha_1} \qquad \dfrac{\dfrac{\rho \vdash x : \text{list}(\alpha_1)}{\rho \vdash tl(x) : \text{list}(\alpha_1)} \qquad \rho \vdash y : \text{list}(\alpha_1)}{\rho \vdash append(tl(x), y) : \text{list}(\alpha_1)}}{\rho \vdash cons(hd(x), append(tl(x), y)) : \text{list}(\alpha_2)}$$

Example Type Inference

fun *append*(*x*, *y*) = **if** *null*(*x*) **then** *y*

else *cons*(*hd*(*x*), *append*(*tl*(*x*), *y*))

The type of *append* : $\forall \sigma, \tau, \varphi.\ (\sigma \times \tau) \rightarrow \varphi$ is:

$\sigma = \text{list}(\alpha)$

$\tau = \varphi = \text{list}(\alpha)$

Hence,

append : $\forall \alpha.(\text{list}(\alpha) \times \text{list}(\alpha)) \rightarrow \text{list}(\alpha)$

Example Type Inference

$$\frac{}{\rho \vdash \mathit{append}([1, 2], [3]) : \tau} \quad \Rightarrow \quad \frac{\rho \vdash ([1, 2],[3]) : \mathrm{list}(\alpha) \times \mathrm{list}(\alpha)}{\rho \vdash \mathit{append}([1, 2], [3]) : \mathrm{list}(\alpha)}$$

$\tau = \mathrm{list}(\alpha)$

$\alpha = \mathrm{integer}$

$$\frac{}{\rho \vdash \mathit{append}([1], [\text{“a”}]) : \tau} \quad \Rightarrow \quad \frac{\rho \vdash ([1],[\text{“a”}]) : \mathrm{list}(\alpha) \times \mathrm{list}(\alpha)}{\rho \vdash \mathit{append}([1], [\text{“a”}]) : \mathrm{list}(\alpha)}$$

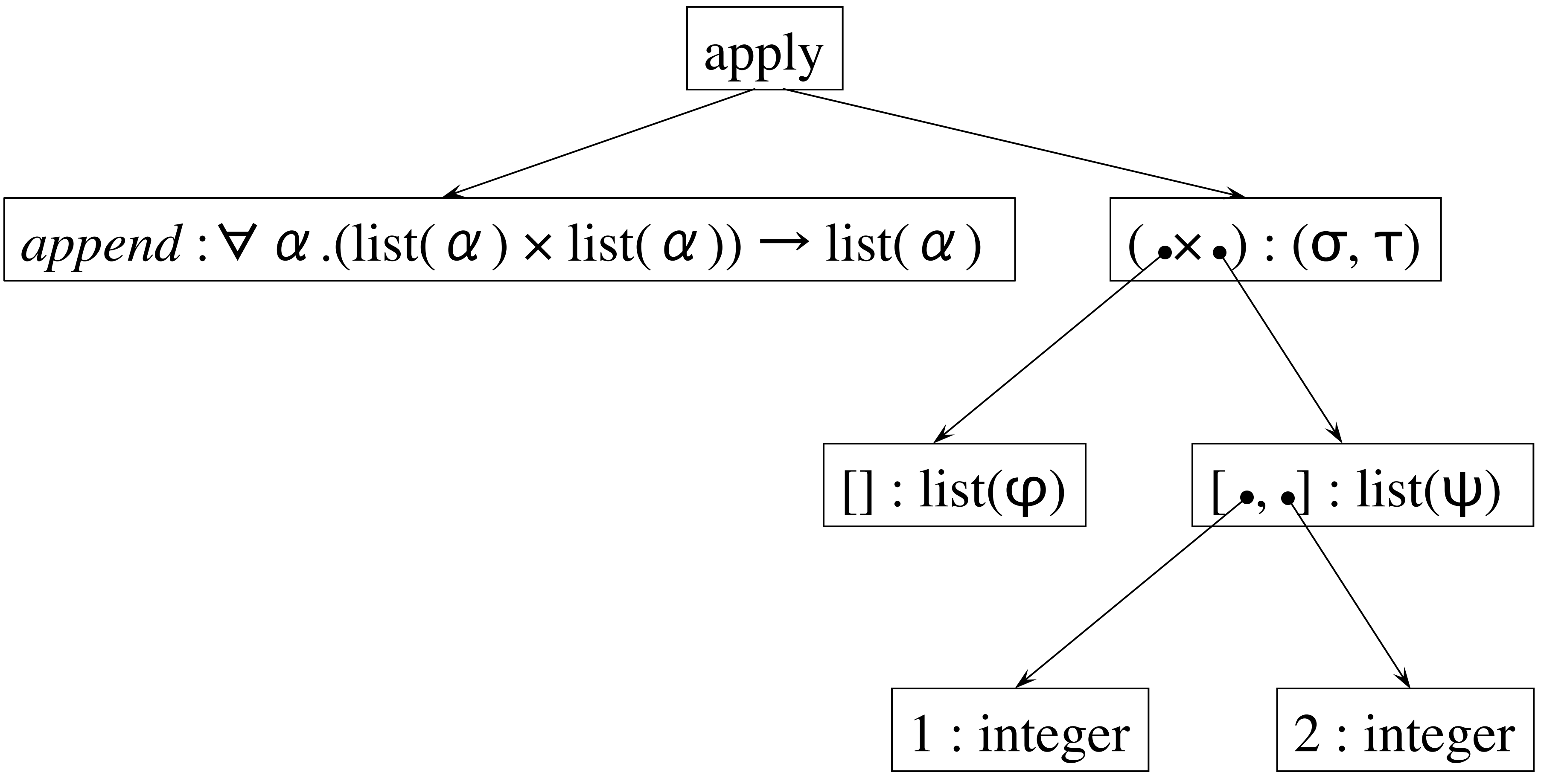
Type error

Type Inference: Substitutions, Instances, and Unification

- The use of a paper-and-pencil post system for type checking/inference involves *substitution*, *instantiation*, and *unification*
- Similarly, in the type inference algorithm, we *substitute* type variables by types to create type *instances*
- A substitution S is a *unifier* of two types t_1 and t_2 if $S(t_1) = S(t_2)$

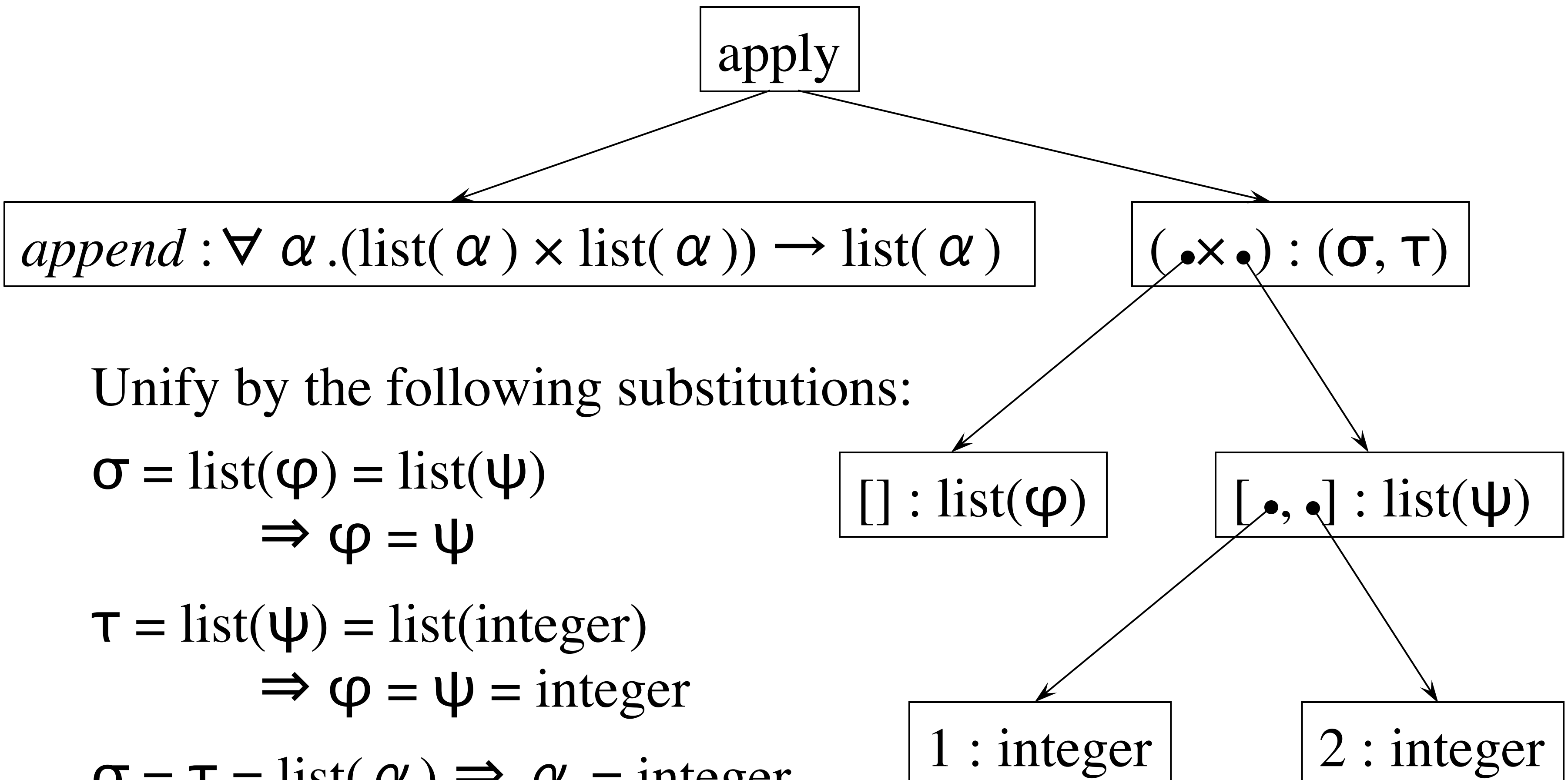
Unification

An AST representation of *append*([], [1, 2])



Unification

An AST representation of *append*([], [1, 2])



Unify by the following substitutions:

$$\sigma = \mathrm{list}(\varphi) = \mathrm{list}(\psi)$$
$$\Rightarrow \varphi = \psi$$

$$\tau = \mathrm{list}(\psi) = \mathrm{list}(\mathrm{integer})$$
$$\Rightarrow \varphi = \psi = \mathrm{integer}$$

$$\sigma = \tau = \mathrm{list}(\alpha) \Rightarrow \alpha = \mathrm{integer}$$