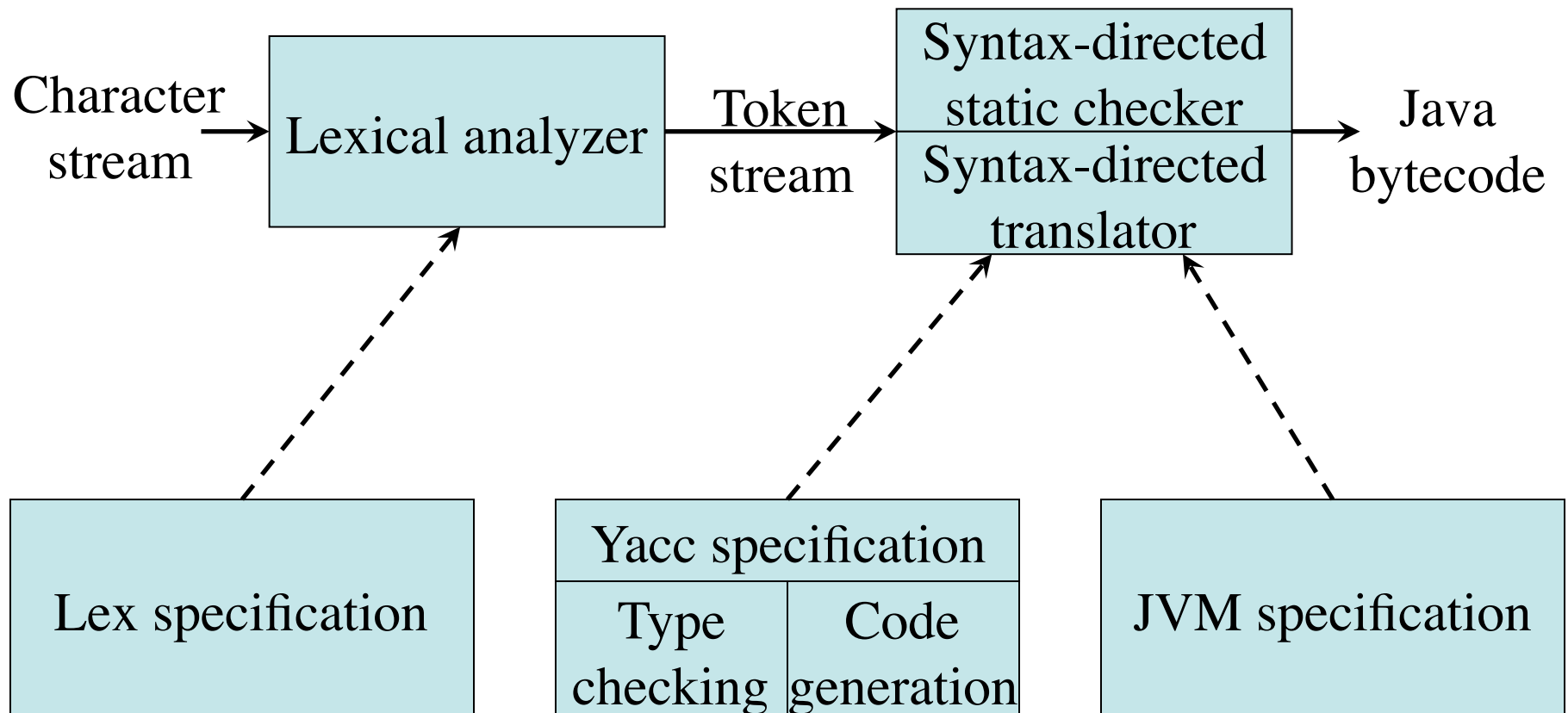


Static Checking and Type Systems

Chapter 6

The Structure of our Compiler Revisited



Static versus Dynamic Checking

- *Static checking*: the compiler enforces programming language's *static semantics*
 - Program properties that can be checked at compile time
- *Dynamic semantics*: checked at run time
 - Compiler generates verification code to enforce programming language's dynamic semantics

Static Checking

- Typical examples of static checking are
 - Type checks
 - Flow-of-control checks
 - Uniqueness checks
 - Name-related checks

Type Checking, Overloading, Coercion, Polymorphism

```
class X { virtual int m(); } *x;
class Y: public X { virtual int m(); } *y;
int op(int), op(float);
int f(float);
int a, c[10], d;

d = c + d;           // FAIL
*d = a;              // FAIL
a = op(d);           // OK: static overloading (C++)
a = f(d);            // OK: coercion of d to float
a = x->m();           // OK: dynamic binding (C++)
vector<int> v;        // OK: template instantiation
```

Flow-of-Control Checks

```
myfunc ()
{ ...
  break; // ERROR
}
```

```
myfunc ()
{ ...
  while (n)
  { ...
    if (i>10)
      break; // OK
  }
}
```

```
myfunc ()
{ ...
  switch (a)
  { case 0:
    ...
      break; // OK
    case 1:
    ...
  }
}
```

Uniqueness Checks

```
myfunc()  
{ int i, j, i; // ERROR  
  ...  
}
```

```
cnufym(int a, int a) // ERROR  
{ ...  
}
```

```
struct myrec  
{ int name;  
};  
struct myrec // ERROR  
{ int id;  
};
```

Name-Related Checks

```
LoopA: for (int I = 0; I < n; I++)
    { ...
      if (a[I] == 0)
        break LoopB; // Java labeled loop
      ...
    }
```

One-Pass versus Multi-Pass Static Checking

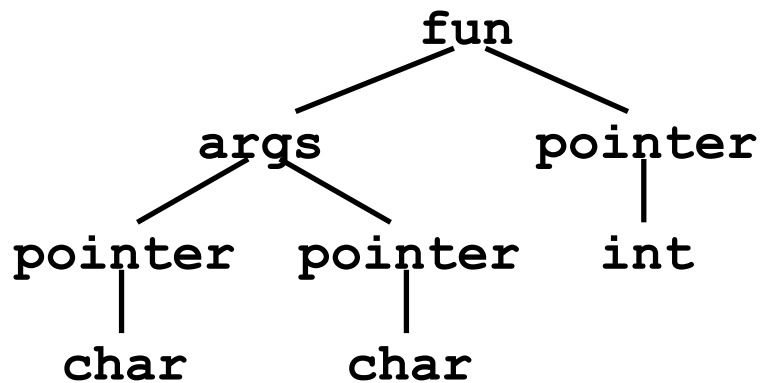
- *One-pass compiler*: static checking in C, Pascal, Fortran, and many other languages is performed in one pass while intermediate code is generated
 - Influences design of a language: placement constraints
- *Multi-pass compiler*: static checking in Ada, Java, and C# is performed in a separate phase, sometimes by traversing a syntax tree multiple times

Type Expressions

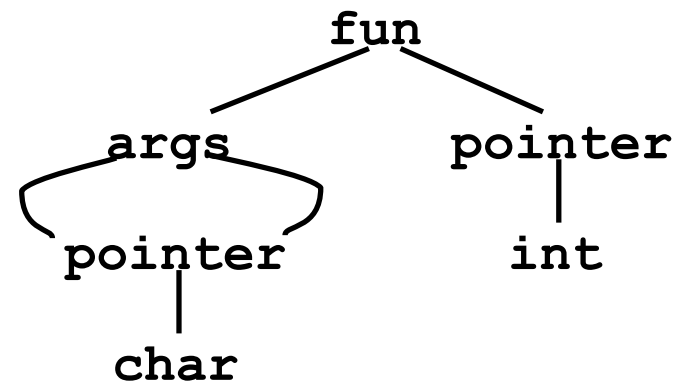
- *Type expressions* are used in declarations and type casts to define or refer to a type
 - *Primitive types*, such as **int** and **bool**
 - *Type constructors*, such as pointer-to, array-of, records and classes, templates, and functions
 - *Type names*, such as typedefs in C and named types in Pascal, refer to type expressions

Graph Representations for Type Expressions

```
int *f(char*,char*)
```



Tree forms

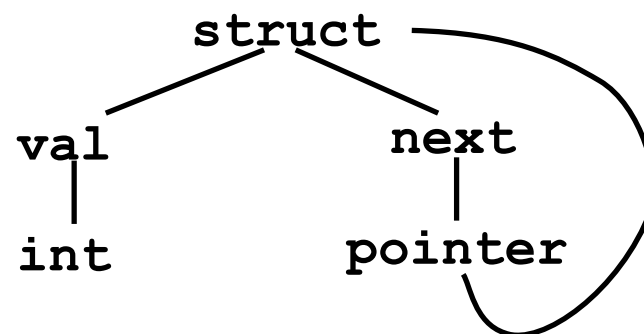


DAGs

Cyclic Graph Representations

Source program

```
struct Node
{ int val;
  struct Node *next;
};
```



Internal compiler representation of
the **Node** type: cyclic graph

Name Equivalence

- Each *type name* is a distinct type, even when the type expressions that the names refer to are the same
- Types are identical only if names match
- Used by Pascal (inconsistently)

```
type link = ^node;
```

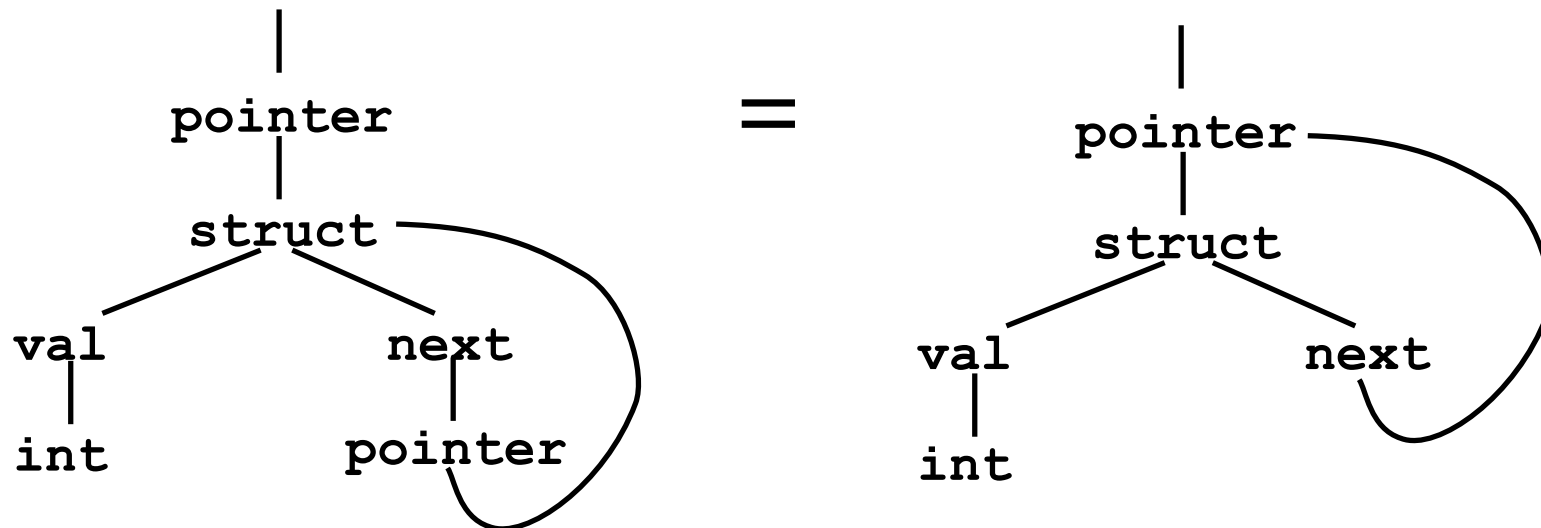
```
var next : link;  
    last : link;  
    p : ^node;  
    q, r : ^node;
```

With name equivalence in Pascal:

```
p ≠ next  
p ≠ last  
p = q = r  
next = last
```

Structural Equivalence of Type Expressions

- Two types are the same if they are *structurally identical*
- Used in C/C++, Java, C#



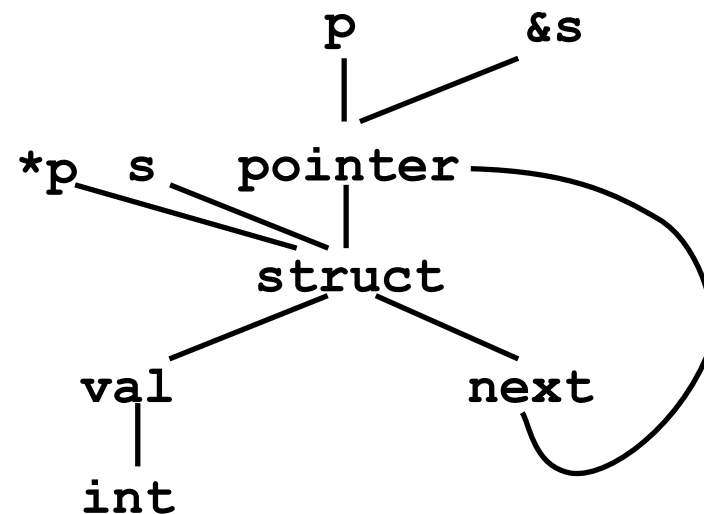
Structural Equivalence of Type Expressions (cont'd)

- Two structurally equivalent type expressions have the same pointer address when constructing graphs by sharing nodes

```

struct Node
{ int val;
  struct Node *next;
};
struct Node s, *p;
p = &s; // OK
*p = s; // OK
p = s;  // ERROR

```



Constructing Type Graphs

Type ***mkint()**

construct int node if not already
constructed

Type ***mkarr(Type*, int)**

construct array-of-type node
if not already constructed

Type ***mkptr(Type*)**

construct pointer-of-type node
if not already constructed

Syntax-Directed Definitions for Constructing Type Graphs

```

%union
{ Symbol *sym;
  int num;
  Type *typ;
}
%token INT
%token <sym> ID
%token <int> NUM
%type <typ> type
%%
decl : type ID          { addtype($2, $1); }
     | type ID '[' NUM ']' { addtype($2, mkarr($1, $4)); }
     ;
type : INT              { $$ = mkint(); }
     | type '*'         { $$ = mkptr($1); }
     | /* empty */     { $$ = mkint(); }
     ;

```

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

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

$$\frac{\rho(v) = \tau \quad \rho \vdash e : \tau}{\rho \vdash v := e : \text{void}}$$

$$\frac{\rho \vdash e_1 : \text{integer} \quad \rho \vdash e_2 : \text{integer}}{\rho \vdash e_1 + e_2 : \text{integer}}$$

Type judgments

$$e : \tau$$

where e is an expression and τ is a type

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

Type System Example

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

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

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

type checking = theorem proving

The proof that $\mathbf{x} := \mathbf{y} + \mathbf{2}$ is typed correctly:

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

A Simple Language Example

$P \rightarrow D ; S$

$D \rightarrow D ; D$

| **id** : T

$T \rightarrow$ **boolean**

| **char**

| **integer**

| **array** [**num**] of T

| $\wedge T$

$S \rightarrow$ **id** := E

| **if** E **then** S

| **while** E **do** S

| $S ; S$

$E \rightarrow$ **true**

| **false**

| **literal**

| **num**

| **id**

| E **and** E

| $E + E$

| $E [E]$

| $E \wedge$

Pointer to T

Pascal-like pointer
dereference operator

Simple Language Example: Declarations

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

Parametric types:
type constructor



Simple Language Example: Checking Statements

$$\frac{\rho(v) = \tau \quad \rho \vdash e : \tau}{\rho \vdash v := e : \text{void}}$$

$S \rightarrow \mathbf{id} := E \{ S.\text{type} := \mathbf{if} \mathbf{id.type} = E.\text{type} \mathbf{then} \text{void} \mathbf{else} \text{type_error} \}$

Note: the type of **id** is determined by scope's environment:
 $\mathbf{id.type} = \text{lookup}(\mathbf{id.entry})$

Simple Language Example: Statements (cont'd)

$$\frac{\rho \vdash e : \textit{boolean} \quad \rho \vdash s : \tau}{\rho \vdash \textbf{while } e \textbf{ do } s : \tau}$$

$S \rightarrow \textbf{while } E \textbf{ do } S_1 \quad \{ S.\textit{type} := \textbf{if } E.\textit{type} = \textit{boolean} \textbf{ then } S_1.\textit{type} \\ \textbf{else } \textit{type_error} \}$

Simple Language Example: Checking Expressions

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

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

Simple Language Example: Checking Expressions (cont'd)

$$\frac{\rho \vdash e_1 : integer \quad \rho \vdash e_2 : integer}{\rho \vdash e_1 + e_2 : integer}$$

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

Simple Language Example: Checking Expressions (cont'd)

$$\frac{\rho \vdash e_1 : \textit{boolean} \quad \rho \vdash e_2 : \textit{boolean}}{\rho \vdash e_1 \mathbf{and} e_2 : \textit{boolean}}$$

$E \rightarrow E_1 \mathbf{and} E_2 \{ E.\textit{type} := \mathbf{if} E_1.\textit{type} = \textit{boolean} \mathbf{and} E_2.\textit{type} = \textit{boolean}$
 $\mathbf{then} \textit{boolean} \mathbf{else} \textit{type_error} \}$

Simple Language Example: Checking Expressions (cont'd)

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

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

Simple Language Example: Checking Expressions (cont'd)

$$\frac{\rho \vdash e : \textit{pointer}(\tau)}{\rho \vdash e^\wedge : \tau}$$

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

A Simple Language Example: Functions

$$T \rightarrow T \rightarrow T$$

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

$$T \rightarrow T_1 \rightarrow T_2 \{ T.type := function(T_1.type, T_2.type) \}$$


Parametric type:
type constructor

Simple Language Example: Checking Function Invocations

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

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

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 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:

$$\mathbf{fun} \text{ length}(x) = \mathbf{if} \text{ null}(x) \mathbf{then} 0 \mathbf{else} \text{ length}(\text{tl}(x)) + 1$$

where

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

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

and the return value is 0 or $\text{length}(\text{tl}(x)) + 1$, thus

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

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:

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

The type of $length(["a", "b"])$ is inferred by

$$\frac{\rho \vdash length : \forall \alpha . list(\alpha) \rightarrow integer \quad \rho \vdash ["a", "b"] : list(string)}{\rho \vdash length(["a", "b"]) : integer}$$

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

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

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

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

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

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

Example Type Inference

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

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

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

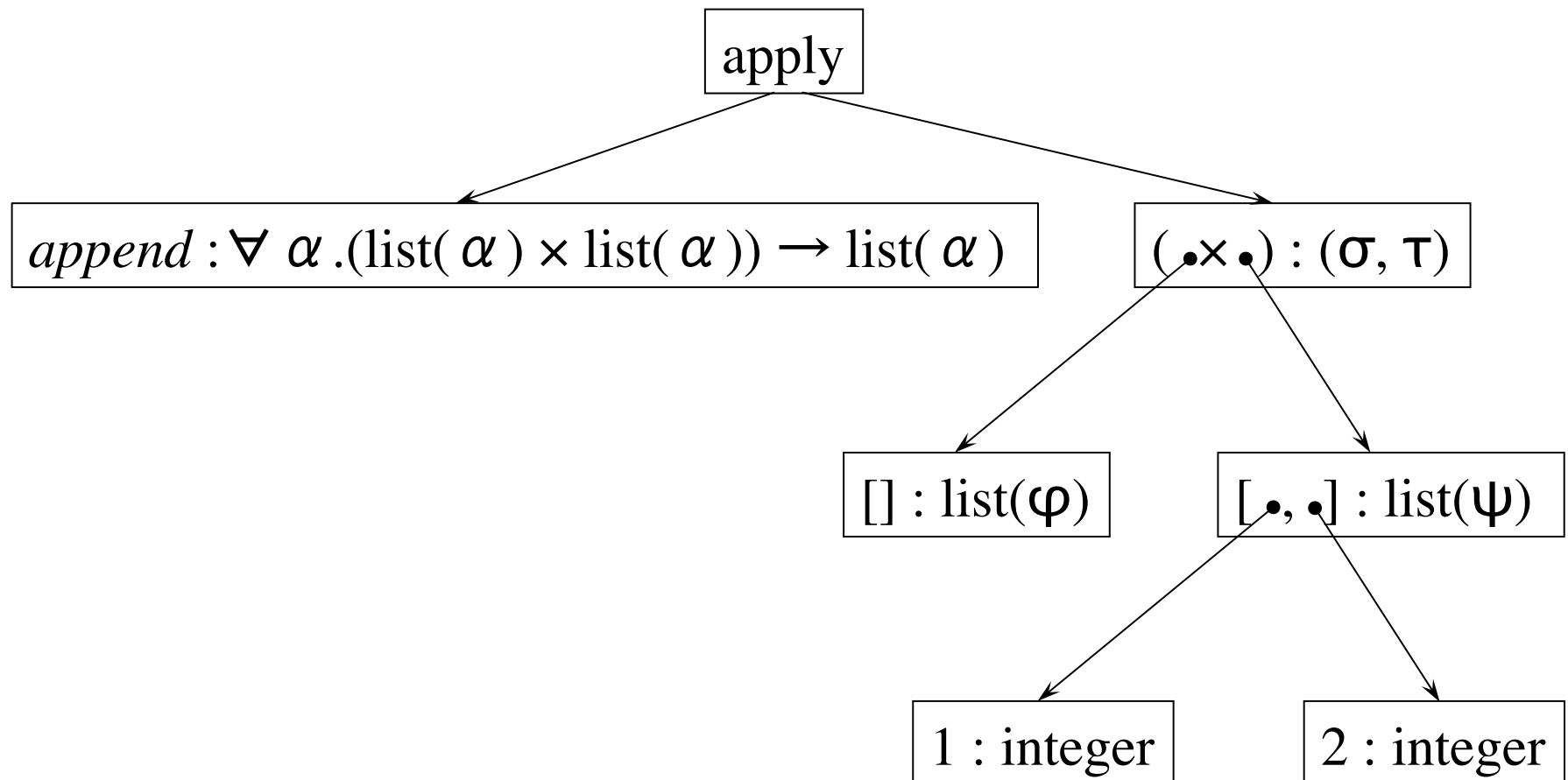
Type error

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 $append([], [1, 2])$

