Type Checking
Type as a Synthesized Attribute

A. Pnueli

• Sufficient for Pascal and C.

end if

Expr(„incompatible types for +“, Expr₁)

Etype(Expr₁) = Any-Type

else

Etype(Expr₁) = Etype(Expr₂)

if Etype( Term ) = Etype( Term )

Expr :: = Expr + Term

169

Honors Compilers, NYU, Spring, 2007
A. Pnueli

Operations on Composite Types

Indexed Component ::= Prex(Expr)

Etype(Indexed Component) = Component

Typ := Etype(Prex)

Etype(Indexed Component) ::= Any–Type

if not is Array Type(A Typ) then
  Error("expect array type in indexed component", Prex)
elsif Etype(Expr) = Index–Type(A Typ) then
  Error("wrong index type", Expr)
elsif Etype(Expr) ≠ Index–Type(A Typ) then
  Error("expect array type in indexed component", Prex)
endif

if not is Array–Type(A Typ) then
  Etype(Indexed–Component) ::= Any–Type
  A Typ := Etype(Prex)
endif

Indexed–Component := Prex(Expr)
Type checking is a set of rules to compute the types of all expressions in the program, and check their compatibility.

**Type Expressions**

- **TypeExpr** ::= access TypeExpr
- **TypeExpr** ::= record (...)
- **TypeExpr** ::= array(TypeExpr) TypeExpr
- **TypeExpr** ::= TypeName
- **TypeExpr** ::= Prim-Type
- **Type constructions**: array, record, access, function.
- **Primitive types**: boolean, float, integer.

**Type Expressions**
Product Types and Functions

A function that takes an argument of type Expr_1

- \text{merge}: \text{array(int)} \times \text{array(int)} \rightarrow \text{real}
- \text{real merge(int}[ \text{a}, \text{int}[ \text{b}])
- \text{Distance}: \text{character} \times \text{character} \rightarrow \text{integer}
- \text{function Distance(C1,C2:character) return integer}

- \text{and returns a value of type Type_2-Expr_2}
- \text{A function that takes an argument of type Type_1-Expr_1}

- Type_1-Expr_1 :: Type_2-Expr_2 \rightarrow \text{Type-Expr_2}
- \text{Intuitively, a pair of types}

- Type_1-Expr_1 :: Type_2-Expr_1 \times \text{Type-Expr_1}
TypeChecking and Type Equivalence

Type correctness can be stated in terms of equivalence of type expressions:

```
arr1 = arr2 // ok in Java: type expressions are equivalent
... 
```

Usually, the rule in languages where a declaration can contain an arbitrary type expression...
The types of Arr1 and Arr2 are not name equivalent. They are, however, structurally equivalent.

Statement `Arr1 := Arr2` is illegal in Ada.

```
Arr2 : anon2
  type anon2 is array(1 .. 10) of integer
  Arr1 : anon1
  type anon1 is array(1 .. 10) of integer
```

Type Checking and Name Equivalence
Type Expressions with Cycles

Lecture 6: Type Checking

A. Pnueli

Honors Compilers, NYU, Spring, 2007

Types are not equivalent:

- C approach: name of type is part of type expression.
- $t_1$ and $t_2$ are equivalent in Algol 68
- $t_1$ and $t_2$ are equivalent in Algol 68

```plaintext
struct cell (int c,
    struct cell* next);
/* struct cell is in type expression */

struct cell (int c,
    struct cell* next);
/* struct cell is in type expression */

struct cell (int c,
    struct cell* next);
/* struct cell is in type expression */

struct cell (int c,
    struct cell* next);
/* struct cell is in type expression */
```
Further complications: user-defined conversions.

- `bool` ← `int`.
- `bit-field` ← `int`.
- `int` ← `(signed/unsigned) char`.
- `char` ← `(signed/unsigned) short`.
- `short` ← `int`.

Otherwise, integral promotions are performed on both:

- `float` ← `double`.
- `double` ← `long double`.
- `long double` ← `double`.

If either operand is of type `long double`, the other is converted to `float`.

For C++ boolean operators:

- `&&` ← `&`.
- `||` ← `|`.
- `|` ← `&`.
- `&` ← `|`.

If language allows coercions, type depends on context; cannot be

**Type Checking and Coercion**
If expression is overloaded, collect set of possible meanings \( sm \).

If context is overloaded, collect set of possible context types \( sc \).

Expression is legal in context if intersection of \( sc \) and \( sm \) is a singleton:

\[
| \text{sc} \cap \text{sm} | = 1
\]

\( sm \) is a singleton.

\( \text{sc} \) is a singleton.

Types \( \text{sc} \). If context is overloaded, collect set of possible context meanings \( sm \). If expression is overloaded, collect set of possible meanings.

Overload Resolution

A. Pnueli

B. Overload Resolution

C++. if multiple interpretations, select the one with smallest number of coercions.

Ada: if multiple interpretations, select predefined numeric operator over user-defined one.

Honors Compilers, NYU, Spring, 2007
P(t(3.14), t(1))

... 

function t(x: float) return boolean
function t(x: integer) return float
function t(x: integer) return integer
function t(x: integer) return integer;

procedure P(x: boolean; z: integer);
procedure P(x: float; y: float);
procedure P(x: integer; y: integer);

Overloaded Context: Procedure Call

A Pnueli
Two-pass Type Resolution

A. Pnueli

Two-pass Type Resolution

A. Pnueli

Two-pass Type Resolution

A. Pnueli

Two-pass Type Resolution

A. Pnueli

Two-pass Type Resolution

A. Pnueli

Two-pass Type Resolution

A. Pnueli

Two-pass Type Resolution

A. Pnueli

Two-pass Type Resolution

A. Pnueli

Two-pass Type Resolution

A. Pnueli
procedure resolve_indexed_component(N: Node_id; Typ: Entity_id) is begin
  find unique interpretation of prefix whose component type is Typ
  resolve index using this interpretation
end;

General scheme:

bottom-up: analyze descendants, synthesize local attribute
top-down: disambiguate construct, propagate to descendants
In a language with list primitives (LISP, ML) what is the type expression that describes CAR or HD?

Car: ∀a· list a → list (a * (∀b· list b))

Tail: ∀a· list a → list a

Head: ∀a· list a

A type variable is universally quantified; the expression is valid for any instantiation of the variable. In ML notation:

- Constructor (::): list (list a)
- Head (hd): ∀a· list a
- Tail (tl): ∀a· list a

A type variable is universally quantified; the expression is valid for any instantiation of the variable. In ML notation:

Car: ∀a· list a → list a → ∀b· list b

Honors Compilers, NYU, Spring 2007
A polymorphic function:

\[
\text{fun } \text{map}(f \text{lis}) = \begin{cases} 
\text{null} & \text{if } \text{lis} = \text{null} \\
\text{f}(\text{hd \text{lis}}) :: \text{map}(f \text{tl \text{lis}}) & \text{otherwise}
\end{cases}
\]

This applies the function \(f\) to each element of a list. The result type of the function is not necessarily equal to the element type of the argument list.

\[
\text{fun \text{Len} \text{lis} = \begin{cases} 
0 & \text{if } \text{lis} = \text{null} \\
1 + \text{Len} (\text{tl \text{lis}}) & \text{otherwise}
\end{cases}
\]

Polymorphic Functions
Polymorphic types have the general form $\forall \text{type variable} : T$, where $T$ is a type expression. For example, a function `reverse` may have the type $\forall \text{list} : \text{list} \rightarrow \text{list}$, where $\text{list}$ is a type.

Following is a grammar for a language with polymorphic functions:

$\text{P} ::=$

- $\text{D} ; \text{E}$
- $\text{D} : \text{id} : \text{Q}$
- $\text{Q} ::=$
  - $\forall \text{type variable} : \text{Q}$
- $\text{T} ::=$
  - $\text{unary constructor} (\text{L})$
  - $\text{L} \times \text{L}$
  - $\text{L}$
  - $\forall \text{type variable} : \text{L}$
  - $\exists \text{D} : \text{D} : \text{E}$

In the non-polymorphic case, we have the following type inference rule:

If $\text{E}_1 : \text{type} \forall \text{type variable} : \text{S}_1$ and $\text{E}_2 : \text{S}_2$, then $\text{E}_1 (\text{E}_2) : \text{S}_2$.

For the polymorphic case, this is generalized to:

If $\text{E}_1 : \text{S}_1$ and $\text{E}_2 : \text{S}_2$, then $\text{E}_1 (\text{E}_2) = \text{S}_2$.

where $\text{mgu}(\text{S}_1, \text{S}_2)$ is the most general unifier of $\text{S}_1$ and $\text{S}_2$. 

$mgu(\text{S}_1, \text{S}_2) = \text{S}_2$.
A Simple Example

Consider the simple program:

\[
\begin{align*}
\text{deref} & : \text{pointer} \\
\text{q} & : \text{pointer(\text{pointer(\text{integer})})} \\
\text{deref(\text{deref(\text{q})})} & : \text{i} \quad \text{where} \quad i = \text{pointer(\text{integer})} \\
\text{deref(\text{q})} & : \text{pointer(\text{integer})} \\
\text{deref(\text{deref(\text{q})})} & : \text{o} \quad \text{where} \quad o = \text{integer} \\
\text{deref(\text{deref(\text{q})})} & : \text{integer} \\
\end{align*}
\]

The type reference proceeds as follows:

\[
\begin{align*}
\text{deref(\text{deref(\text{deref(\text{q})})})} & : \text{integer} \\
\text{deref(\text{deref(\text{deref(\text{q})})})} & : \text{b} \\
\text{deref(\text{deref(\text{deref(\text{q})})})} & : \text{a} \\
\end{align*}
\]
Checking Polymorphic Types

The function \texttt{fresh(id, type)} constructs a new type node in which all quantified type variables are renamed by fresh names.

Following is a translation scheme which shows how to infer the types in a polymorphic program as a synthesized attribute:

\[
\begin{align*}
\text{E} :&= \text{E}_1 \ (\text{E}_2) \\
\text{f}_{\text{p}} :&= \text{mkleaf}(\text{newtypevar}) \\
\text{unify}(\text{E}_1, \text{E}_2) :&= \text{mknode}(\otimes, \text{E}_1, \text{E}_2) \\
\text{fresh}(\text{id}, \text{type}) &\quad \leadsto \\
\text{id} :&= \text{fresh}(\text{id}, \text{type}) \\
\text{E}_1, \text{E}_2 :&= \text{fresh}(\text{id}, \text{type}) \\
\text{d} :&= \text{mkleaf}(\text{newtypevar}) \\
\text{E}_1 \ (\text{E}_2) &\quad \leadsto
\end{align*}
\]

Checking Polymorphic Types
Algorithm for Polymorphic Type Inference

A.Pnueli

Algorithm 6: Type Checking

- For a newly encountered (undeclared) variable, introduce a fresh type variable.
- For each occurrence of a polymorphic function, introduce fresh type variables for all bounded.
- For an application of a polymorphic function, type variables for all free.
- For an application $E_1(E_2)$, infer types $s$ and $t$ for $E_1$ and $E_2$. Unify $s$ with $t$. The type of $E_1(E_2)$ is $s$. (after unification).
- For a function definition $\text{fun id}_1(id_2) = E$, assign types $id_2 : s$, $id_1 : !t$ (and fresh type variables). Infer type for $E$. Assume $\alpha$ and evolved into $s$ and $t$ during the inference. The type for $id_1$ is $\alpha$. Assume $\beta$ and fresh type variables. Infer types $\alpha : \beta$. Assign types $\alpha : \beta$. $E = (\beta \beta_1)$.
- For each occurrence of a newly encountered (undeclared) variable, introduce a fresh type variable.
- Applicable for strongly typed languages which do not require the types of all names to be explicitly declared.
- Applicable for polymorphic type inference.
TypeInferenceforthe length Function

Consider the ML function:

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

which can be elaborated into

```
Honors Compilers, NYU, Spring, 2007

A. Pnueli

Lecture 6: Type Checking

Type Inference for the Length Function
```
<table>
<thead>
<tr>
<th>Substitution</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>\texttt{\[x\]} = \mathsf{\_}</code></td>
<td><code>\texttt{\_}</code></td>
</tr>
<tr>
<td><code>\texttt{\[\texttt{\_}\]} = \mathsf{\_}</code></td>
<td><code>\texttt{\_}</code></td>
</tr>
<tr>
<td><code>\texttt{\[\texttt{\_}\]} = \mathsf{\_}</code></td>
<td><code>\texttt{\_}</code></td>
</tr>
</tbody>
</table>

Inference: Type Checking

\[ \text{if} \ (\exists) \ (\exists) \ (\exists) \ (\exists) \]
A.Pnueli

The Limits of Type Inference

Self-application is not typable:

\[ (g' \leftarrow x) = x \]
\[ \text{int} = g' \]

9 applied to itself yields an integer

\[ g' \leftarrow x : g \]

9 is a function

\[ (g' \leftarrow x) = x \]

Self-application is not typable:

\[ (g)g + 1 \]

\[ \text{The Limits of Type Inference} \]