Type Checking

Type Checking is a set of rules to compute the types of all expressions in the program, and check their compatibility.

Expressions::=Expr+Term
if Etype(Expr)=Etype(Term)
then Etype(Expr):=Etype(Expr)
else Etype(Expr):=Any
endif

Sufficient for Pascal and C.

Type Expressions

Operations on Composite Types

Indexed Component::=Prex(Expr)
A Typ:=Etype(Prex)
Etype(Indexed Component):=Any–by default
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)
else Etype(Indexed Component):=Component Type(A Typ)
endif

Type Checking is a set of rules to compute the types of all expressions in the program, and check their compatibility.

Type as a Synthesized Attribute
Type expressions with Cycles

Types are not equivalent:

• C approach: name of type is part of type expression.
• type T1 = record c: integer; p: pointer(T2); end record;
• type T2 = record c: integer; p: pointer(T1); end record;

Intuitively, a pair of types

The types of Arr1 and Arr2 are name equivalent.
Statement Arr1 := Arr2 is illegal in Ada.

They are, however, structurally equivalent.

Type Checking and Type Equivalence

Type correctness can be stated in terms of equivalence.

Types of Arr1 and Arr2 are equivalent in Algol 68.

Types are not equivalent:

• struct cell (int c, struct cell *next);

Type expressions can be stated in terms of equivalence.

Type Checking and Name Equivalence

Statement Arr1 := Arr2 is array(1..10) of integer.

Types Arr1 and Arr2 are name equivalent.
They are, however, structurally equivalent.

Function Distance (C1,C2: Character) return integer

A function that takes an argument of type type-expr1

• type-expr :: = type-expr1 * type-expr2

Likewise the rule in languages where a declaration can
contain an arbitrary type expression

usualy the rule in languages where a declaration can

לרל hausarbeit

• arr1 = arr2

array(int) 

array(int) 

by hand

merge : array(int) 

merge : array(int)

Distinct character * character return integer

A function that takes an argument of type type-expr1

• type-expr :: = type-expr1 * type-expr2

type-expr :: = type-expr1 * type-expr2

Having compiled, NYU, Fall, 2009

174

Honors Compilers, NYU, Fall, 2009

175

Honors Compilers, NYU, Fall, 2009

176

Honors Compilers, NYU, Fall, 2009

177
TypeChecking

If language allows coercions, type depends on context, cannot be purely synthesized.

For C++ boolean operators:

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

Otherwise, if either operand is double, the other is converted to double.

Otherwise, if either is float, the other is converted to float.

Otherwise, if either is short int, the other is converted to short int.

Otherwise, if either is char, the other is converted to char.

Otherwise, integral promotions are performed on both:

For C++ boolean operators:

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

For C++ boolean operators:

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

User-defined conversions.

Further complications:

- Small number of coercions.
- Multiple interpretations, select predefined type.
- ADA: if multiple interpretations, select predefined type.
- C++: if multiple interpretations, selects the one with smallest number of coercions.
- Expression is legal in context if intersection of sc and tyeps = 1.
- Otherwise, if either operand is float, the other is converted to float.
- Otherwise, if either operand is double, the other is converted to double.
- Otherwise, if either operand is int, the other is converted to int.
- Otherwise, if either operand is char, the other is converted to char.
- Otherwise, integral promotions are performed on both:
- Otherwise, if either operand is double, the other is converted to double.
- Otherwise, if either operand is float, the other is converted to float.
- Otherwise, if either operand is int, the other is converted to int.
- Otherwise, if either operand is char, the other is converted to char.

Procedure:

Two-Pass Type Resolution

- Bottom-up (analyze): synthesize candidate types.
- Top-down (resolve): propagate unique context type.

Overloaded Context: Procedure Call

Overload Resolution

numeric operator over user-defined one.

expression is overloaded, select predefined one.

else remove t from pte

then add component: Type(p) to interpretations of N

if there is a compatible ix in Ind then

begin

analyze prefx, collect set of interpretations Ind

analyze idx, collect set of interpretations Ind

analyze prefx, collect set of interpretations Ind

end

forall t in pre loop

analyze prefx-indexed component (N:Node:id) is

• Top-down (resolve): Propagate unique context type.

• Bottom-up (analyze): Synthesize candidate types.

end loop

if there is a compatible ix in Ind then

add component Type(t) to interpretations of N

else remove t from Pre

endif

end loop

analyze prefx, collect set of interpretations Ind

analyze idx, collect set of interpretations Ind

analyze prefx, collect set of interpretations Ind

end
Two-pass Type Resolution (2)

A. Pnueli

Lecture 6: Type Checking

Two-pass Type Resolution (2)

procedure resolve indexed component (N: Node id; Typ: Entity id)

begin
  resolve index using this interpretation
end;

Generalscheme:
bottom-up: analyzedescendants, synthesizelocalattribute
top-down: disambiguatesynthesize localattribute

Honors Compilers, NYU, Fall, 2009

TypeVariablesandPolymorphism

In a language with list primitives (LISP, ML) what is the type expression that describes CAR or hd?

Informally, given a list of components of any kind, CAR yields a value of that kind.

Top down: disambiguatesynthesize localattribute
Polymorphic Functions

Polymorphic types have the general form

\[ \forall \text{type variable: } T \]

A type variable is universally quantified; the expression is valid for any instantiation of the variable. In ML, type variables are like any other variables: in ML, a type variable is universally quantified; the expression is valid for any instantiation of the variable.

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

\[ \begin{array}{c}
\text{Constructor} (\text{::}) \rightarrow \text{a} \\
\text{Tail} (\text{il}) \rightarrow \text{il} \\
\text{Head} (\text{il}) \rightarrow \text{il}
\end{array} \]

Notation:

\[ \begin{array}{c}
\text{Car: } \text{il} \rightarrow \text{whatever} \\
\text{List of whatever: } \text{il} \rightarrow \text{whatever}
\end{array} \]

Informally, given a list of components of any kind, CAR or hd yields a value of that kind.

In a language without list primitives (LISP, ML) what is the type of CAR or hd?

In ML, type variables are like any other variables: in ML, a type variable is universally quantified; the expression is valid for any instantiation of the variable.
Type Checking

A. Pnueli

Consider the simple program:

```
deref : 8 : pointer (
    !
    :
    q : pointer (pointer (integer (x : integer)) )
```

The type inference proceeds as follows:

```
deref : pointer (
    !
    q : pointer (pointer (integer (x : integer)) )
```

```
deref (deref (q)) : i where i = pointer (integer (x : integer))
```

```
deref (q) : pointer (integer (x : integer))
```

```
deref (deref (q)) : o where o = integer
```

```
deref (deref (q)) : integer
```

Honors Compilers, NYU, Fall, 2009

Checking Polymorphic Types

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

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

For a newly encountered (undeclared) variable, introduce a fresh type variable.

Type variables for all bounded.

For each occurrence of a polymorphic function, introduce fresh

Unify $s$ with $t$. The type of $E (F; x)$ is $s$ (after unification).

For an application $E_1 (E_2)$, infer types $s$ and $t$ for $E_1$ and $E_2$.

```
E := E_1 (E_2)
```

The type for $id_1$ is $s$!$t$.

```
for (E) id_2 := E_1, E_2
```

```
Assume $a$ and $b$ evolved into $s$ and $t$ during the inference. The
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
Inferring the Types

Expression: $\text{length}(\text{tl}(x)) = \text{integer}$

Substitution: $\text{integer}$

The Limitsof TypeInference

Honors Compilers, NYU, Fall, 2009

Lecture 6: TypeChecking

A. Pnueli