Type Checking

Type expressions in the program, and check their compatibility.

Type Checking is a set of rules to compute the types of all

- Type-Expr ::＝ access (Type-Expr)
- Type-Expr ::＝ record (Expr)
- Type-Expr ::＝ array (Type-Expr) (Type-Expr)
- Type-Expr ::＝ type-Name
- Type-Expr ::＝ Prim-Name
- Type-Expr ::＝ Prim-Name
- Type-Expr ::＝ built From

Type Expressions

Sufficient for Pascal and C.

- end if
- Error ("incompatible types for +", Expr)
- Error ("wrong index type", Expr)
- Error ("expect array type in indexed component", Prex)
- Error ("wrong index type", Expr)
- Error ("array type in indexed component", Prex)
- if A-Expr = Index Type (A-Typ)
- endif

Operations on Composite Types

Indexed Component ::＝ Prex (Expr)

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

Type as a Synthesized Attribute

Lecture 6: Type Checking

NYU, Spring, 2007
TypeChecking

TypeChecking and Type Equivalence

Typecorrectness can bestated intermsoftequivalence oftypeexpressions:

int [] arr1://arr1:

array(int)

int [] arr2://arr2:

array(int)

arr1=arr2

//okinJava:typeexpressionsareequivalent

TypeChecking and Name Equivalence

Statement Arr1 := Arr2 is illegal in Ada.

The types of Arr1 and Arr2 are not nameequivalent.

They are, however structurallyequivalent.

TypeExpressions with Cycles

They types are not equivalent:
• C approach: nameof type is part of typeexpression.

• T1 and T2 are equivalent in Algol68.

T1 = record c: integer; p: pointer(T2); end record;

T2 = record c: integer; p: pointer(T1); end record;

Type Checking and Name Equivalence

TypeExpressions with Cycles

They types are not equivalent:
• C approach: name of type is part of typeexpression.

• T1 and T2 are equivalent in Algol68.

T1 = record c: integer; p: pointer(T2); end record;

T2 = record c: integer; p: pointer(T1); end record;

Type Checking and Name Equivalence

TypeExpressions with Cycles

They types are not equivalent:
• C approach: name of type is part of typeexpression.

• T1 and T2 are equivalent in Algol68.

T1 = record c: integer; p: pointer(T2); end record;

T2 = record c: integer; p: pointer(T1); end record;

Type Checking and Name Equivalence

TypeExpressions with Cycles

They types are not equivalent:
• C approach: name of type is part of typeexpression.

• T1 and T2 are equivalent in Algol68.

T1 = record c: integer; p: pointer(T2); end record;

T2 = record c: integer; p: pointer(T1); end record;

Type Checking and Name Equivalence

TypeExpressions with Cycles

They types are not equivalent:
• C approach: name of type is part of typeexpression.

• T1 and T2 are equivalent in Algol68.

T1 = record c: integer; p: pointer(T2); end record;

T2 = record c: integer; p: pointer(T1); end record;

Type Checking and Name Equivalence

TypeExpressions with Cycles

They types are not equivalent:
• C approach: name of type is part of typeexpression.

• T1 and T2 are equivalent in Algol68.

T1 = record c: integer; p: pointer(T2); end record;

T2 = record c: integer; p: pointer(T1); end record;

Type Checking and Name Equivalence

TypeExpressions with Cycles

They types are not equivalent:
• C approach: name of type is part of typeexpression.

• T1 and T2 are equivalent in Algol68.

T1 = record c: integer; p: pointer(T2); end record;

T2 = record c: integer; p: pointer(T1); end record;

Type Checking and Name Equivalence

TypeExpressions with Cycles

They types are not equivalent:
• C approach: name of type is part of typeexpression.

• T1 and T2 are equivalent in Algol68.

T1 = record c: integer; p: pointer(T2); end record;

T2 = record c: integer; p: pointer(T1); end record;

Type Checking and Name Equivalence

TypeExpressions with Cycles

They types are not equivalent:
• C approach: name of type is part of typeexpression.

• T1 and T2 are equivalent in Algol68.

T1 = record c: integer; p: pointer(T2); end record;

T2 = record c: integer; p: pointer(T1); end record;

Type Checking and Name Equivalence

TypeExpressions with Cycles

They types are not equivalent:
• C approach: name of type is part of typeexpression.

• T1 and T2 are equivalent in Algol68.

T1 = record c: integer; p: pointer(T2); end record;

T2 = record c: integer; p: pointer(T1); end record;

Type Checking and Name Equivalence

TypeExpressions with Cycles

They types are not equivalent:
• C approach: name of type is part of typeexpression.

• T1 and T2 are equivalent in Algol68.

T1 = record c: integer; p: pointer(T2); end record;

T2 = record c: integer; p: pointer(T1); end record;

Type Checking and Name Equivalence

TypeExpressions with Cycles

They types are not equivalent:
• C approach: name of type is part of typeexpression.

• T1 and T2 are equivalent in Algol68.

T1 = record c: integer; p: pointer(T2); end record;

T2 = record c: integer; p: pointer(T1); end record;

Type Checking and Name Equivalence

TypeExpressions with Cycles

They types are not equivalent:
• C approach: name of type is part of typeexpression.

• T1 and T2 are equivalent in Algol68.

T1 = record c: integer; p: pointer(T2); end record;

T2 = record c: integer; p: pointer(T1); end record;

Type Checking and Name Equivalence

TypeExpressions with Cycles

They types are not equivalent:
• C approach: name of type is part of typeexpression.

• T1 and T2 are equivalent in Algol68.

T1 = record c: integer; p: pointer(T2); end record;

T2 = record c: integer; p: pointer(T1); end record;

Type Checking and Name Equivalence

TypeExpressions with Cycles

They types are not equivalent:
• C approach: name of type is part of typeexpression.

• T1 and T2 are equivalent in Algol68.

T1 = record c: integer; p: pointer(T2); end record;

T2 = record c: integer; p: pointer(T1); end record;

Type Checking and Name Equivalence

TypeExpressions with Cycles

They types are not equivalent:
• C approach: name of type is part of typeexpression.

• T1 and T2 are equivalent in Algol68.

T1 = record c: integer; p: pointer(T2); end record;

T2 = record c: integer; p: pointer(T1); end record;

Type Checking and Name Equivalence

TypeExpressions with Cycles

They types are not equivalent:
• C approach: name of type is part of typeexpression.

• T1 and T2 are equivalent in Algol68.

T1 = record c: integer; p: pointer(T2); end record;

T2 = record c: integer; p: pointer(T1); end record;

Type Checking and Name Equivalence

TypeExpressions with Cycles

They types are not equivalent:
• C approach: name of type is part of typeexpression.

• T1 and T2 are equivalent in Algol68.

T1 = record c: integer; p: pointer(T2); end record;

T2 = record c: integer; p: pointer(T1); end record;

Type Checking and Name Equivalence

TypeExpressions with Cycles

They types are not equivalent:
• C approach: name of type is part of typeexpression.

• T1 and T2 are equivalent in Algol68.

T1 = record c: integer; p: pointer(T2); end record;

T2 = record c: integer; p: pointer(T1); end record;

Type Checking and Name Equivalence

TypeExpressions with Cycles

They types are not equivalent:
• C approach: name of type is part of typeexpression.

• T1 and T2 are equivalent in Algol68.

T1 = record c: integer; p: pointer(T2); end record;

T2 = record c: integer; p: pointer(T1); end record;

Type Checking and Name Equivalence

TypeExpressions with Cycles

They types are not equivalent:
• C approach: name of type is part of typeexpression.

• T1 and T2 are equivalent in Algol68.

T1 = record c: integer; p: pointer(T2); end record;

T2 = record c: integer; p: pointer(T1); end record;

Type Checking and Name Equivalence

TypeExpressions with Cycles

They types are not equivalent:
• C approach: name of type is part of typeexpression.

• T1 and T2 are equivalent in Algol68.

T1 = record c: integer; p: pointer(T2); end record;

T2 = record c: integer; p: pointer(T1); end record;

Type Checking and Name Equivalence

TypeExpressions with Cycles

They types are not equivalent:
• C approach: name of type is part of typeexpression.

• T1 and T2 are equivalent in Algol68.

T1 = record c: integer; p: pointer(T2); end record;

T2 = record c: integer; p: pointer(T1); end record;

Type Checking and Name Equivalence

TypeExpressions with Cycles

They types are not equivalent:
• C approach: name of type is part of typeexpression.

• T1 and T2 are equivalent in Algol68.

T1 = record c: integer; p: pointer(T2); end record;

T2 = record c: integer; p: pointer(T1); end record;

Type Checking and Name Equivalence

TypeExpressions with Cycles

They types are not equivalent:
• C approach: name of type is part of typeexpression.

• T1 and T2 are equivalent in Algol68.

T1 = record c: integer; p: pointer(T2); end record;

T2 = record c: integer; p: pointer(T1); end record;

Type Checking and Name Equivalence

TypeExpressions with Cycles

They types are not equivalent:
• C approach: name of type is part of typeexpression.

• T1 and T2 are equivalent in Algol68.

T1 = record c: integer; p: pointer(T2); end record;

T2 = record c: integer; p: pointer(T1); end record;

Type Checking and Name Equivalence

TypeExpressions with Cycles

They types are not equivalent:
• C approach: name of type is part of typeexpression.

• T1 and T2 are equivalent in Algol68.

T1 = record c: integer; p: pointer(T2); end record;

T2 = record c: integer; p: pointer(T1); end record;

Type Checking and Name Equivalence

TypeExpressions with Cycles

They types are not equivalent:
• C approach: name of type is part of typeexpression.

• T1 and T2 are equivalent in Algol68.

T1 = record c: integer; p: pointer(T2); end record;

T2 = record c: integer; p: pointer(T1); end record;

Type Checking and Name Equivalence

TypeExpressions with Cycles

They types are not equivalent:
• C approach: name of type is part of typeexpression.

• T1 and T2 are equivalent in Algol68.

T1 = record c: integer; p: pointer(T2); end record;

T2 = record c: integer; p: pointer(T1); end record;

Type Checking and Name Equivalence

TypeExpressions with Cycles

They types are not equivalent:
• C approach: name of type is part of typeexpression.

• T1 and T2 are equivalent in Algol68.

T1 = record c: integer; p: pointer(T2); end record;

T2 = record c: integer; p: pointer(T1); end record;

Type Checking and Name Equivalence

TypeExpressions with Cycles

They types are not equivalent:
• C approach: name of type is part of typeexpression.

• T1 and T2 are equivalent in Algol68.

T1 = record c: integer; p: pointer(T2); end record;

T2 = record c: integer; p: pointer(T1); end record;
Type Checking and Coercion

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 operand is `float`, the other is converted to `float`.
- Otherwise, if either operand is `char` (un)signed, the other is converted to `int`.
- Otherwise, if either operand is `long` double, the other is converted to `long double`.
- Otherwise, if either operand is `bool`, the other is converted to `bool`.

Type Checking and Coercion

Overloaded Context: Procedure Call

```latex
\text{procedure} \text{P}(x: \text{integer}; y: \text{oat});
\text{procedure} \text{P}(x: \text{oat}; y: \text{oat});
\text{procedure} \text{P}(x: \text{boolean}; z: \text{integer});
```

Two-pass Type Resolution

Bottom-up (analyze): synthesize candidate types.

Top-down (resolve): propagate unique context types.

- Ada: if multiple interpretations, select predefined one.
- C++: if multiple interpretations, select predefined one.
- sm is a singleton: \( \{ \text{sm} \} = \{ t \}. \)
- Otherwise, if either operand is `long double`, the other is converted to `long double`.
- Otherwise, if either operand is `char` (un)ssigned, the other is converted to `int`.
- 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 `char` (un)ssigned, the other is converted to `int`.
- Otherwise, if either operand is `long` double, the other is converted to `long double`.
- Otherwise, if either operand is `bool`, the other is converted to `bool`.

Further complications: user-defined conversions.

- For C++ boolean operators:
  - Multiply `unsigned`.
- If language allows coercions, type depends on context, cannot be purely synthesized.
Two-pass Type Resolution (2)

A. Pnueli

Type Variables and Polymorphism

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

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

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

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

Polymorphic Types

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

Polymorphic Functions

Polyomorphicc types have the general form

\[ \forall type\ variable : T \], where \( T \) is a type expression.

For example, a function list reverse may have the type

\[ \forall : list (\forall : list) \rightarrow list \] , implying that it reverses a list of "anything".

Following is a grammar for a language with polymorphic functions:

\[
P ::= D ; E \\
D ::= D ; j id : Q \\
Q ::= \exists \; type\ variable : Q j T \\
T ::= T ; j T \mid j (T) \\
E ::= E (E) j E ; E j id
\]

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

\[
\begin{align*}
\text{If} & \ E_1 : s ! t \text{ and } E_2 : s, \text{then } E_1 (E_2) : t. \\
\end{align*}
\]

For the polymorphic case, this is generalized to:

\[
\begin{align*}
\text{If} & \ E_1 : \forall : s_1 ! t \text{ and } E_2 : s_2, \text{then } E_1 (E_2) : t \left[ \text{mgu} (s_1; s_2) \right]. \\
\end{align*}
\]

where \( \text{mgu} (s_1; s_2) \) is the most general unifier of \( s_1 \) and \( s_2 \).

If \( E_1 : \forall : s_1 ! t \) and \( E_2 : s_2 \), then \( E_1 (E_2) : s_2 ! t \).

For the polymorphic case, this is generalized to:

\[
\begin{align*}
\text{If} & \ E_1 : \forall : s_1 ! t \text{ and } E_2 : s_2, \text{then } E_1 (E_2) : s_2 ! t. \\
\end{align*}
\]
Consider the simple program:

deref : 8 : pointer
q : pointer (pointer (integer))

\[ \text{deref (deref (q))} \]

The type reference proceeds as follows:

\[ \text{deref : pointer (pointer (integer))} \]
\[ \text{q : pointer (pointer (integer))} \]
\[ \text{deref : integer} \]
\[ \text{deref (q) : integer} \]
\[ \text{deref (deref (q)) : integer} \]

Honors Compilers, NYU, Spring, 2007
Lecture 6: Type Checking

A. Pnueli

Inferring the Types

Expression

Substitution

The Limits of Type Inference

Inerring the Types

Self-application is not typable:

9 applied to itself yields an integer

9 = integer

Self-application is not typable:

\[ (g)g + 1 = \text{integer} \]

\[ g \rightarrow \text{integer} \]

\[ g \in \text{integer} \]

Circular definition (not typable):

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

\[ \text{integer} = \text{integer} \]

\[ \text{integer} = x \]

\[ \text{integer} = (x) \]

\[ \text{integer} = \text{integer} \]

\[ \text{integer} = \text{integer} \]

\[ \text{integer} = \text{integer} \]

\[ \text{integer} = \text{integer} \]

\[ \text{integer} = \text{integer} \]