Announcements

• Office hours today at 4:00pm.

• Project 1 has been graded. Grades will be released today.

• [Project 2 extension needed]? Current due date: Friday April 8.

• Fifth homework will be posted by tomorrow.

• Roadmap: Type systems, logic programming (Prolog), parallel programming
**Typing and Type Systems**

Read Scott: Chapter 7.1 - 7.2; ALSU Chapter 6.5

**type:**

| A set of values and meaningful operations on them. |

Types provide semantic “sanity checks” (consistency checks). Types help identify:

- errors, if an operator is applied to an incompatible operand.
  - dereferencing of a non-pointer
  - adding a function name to something
  - incorrect number of parameters to a procedure
  - …
- which operation to use for overloaded names and operators, or what type coercion to use (e.g.: 3.0 + 1)
- identification of polymorphic functions. Such functions can be executed with arguments of several types (e.g.: list of objects of unknown type $\alpha$)
Type systems

Each language construct (operator, expression, statement, ...) has a type

**basic types:** integer, real, character, ...

**constructed types:** arrays, records, sets, pointers, functions

A **type system** is a collection of *rules* for assigning *type expressions* to operators, expressions, ... in the program. Type systems are language dependent.

A **type checker** implements the type system, i.e., deduces type expressions for program constructs based on the type inference rules of the type system. The type checker “computes” or “reconstructs” type expressions.
Example type rules

- If both operands of the arithmetic operators of addition, subtraction, and multiplication are of type integer, then the result is of type integer (Pascal definition).

Rule for + (analogue rules for − and ∗):

\[ E \vdash e_1 : integer \quad E \vdash e_2 : integer \]
\[ \overline{\quad E \vdash (e_1 + e_2) : integer \quad} \]

where \( E \) is a type environment that maps constants and variables to their types.

In combination with the following axiom template in the type system \{c : \alpha\} \vdash c : \alpha
we can now infer that \((2 + 3)\) is of type integer:

\[ E \vdash 2 : integer \quad E \vdash 3 : integer \]
\[ \overline{\quad E \vdash (2 + 3) : integer \quad} \]

where \( E = \{2 : integer, 3 : integer\} \).

In general, type deduction proofs work bottom up.
Example type rules

- The result of the unary \& operator is a pointer to the object referred to by the operand. If the operand is of type “foo”, then the type of the result is a “pointer to foo”. (C and C++ definition)

\[
\begin{align*}
E \vdash e : \alpha \\
\vdash \&e : \text{pointer}(\alpha)
\end{align*}
\]

- Two expressions can only be compared if they have the same types. The result is of type boolean.

\[
\begin{align*}
E \vdash e_1 : \alpha & \quad E \vdash e_2 : \alpha \\
\vdash (e_1 = e_2) : \text{boolean}
\end{align*}
\]

In the examples, integer and \( \alpha \) are type expressions.
Type expressions  

inductive definition

1. A basic type is a type expression. A special basic type, `typeError` will signal an error. A basic type `void` denotes an untyped statement.

2. Since type expressions may be named, a type name is a type expression.

3. Type expressions may contain variables whose values are type expressions (e.g.: useful for languages without type declarations, or polymorphism).

4. A type constructor applied to type expressions is a type expression. Examples:

   (a) arrays
   (b) cartesian products
   (c) records
   (d) pointers
   (e) functions
Type checking

The purpose of type checking is to prevent type errors. Type errors can occur during the execution of a program, for instance, when a function is applied with an argument of the wrong type or if a variable with a non–function type is called.

“How much” type checking is done is part of the definition of the programming language.

Type checking can be performed

• at compile–time (only)
• at compile–time and program execution time, or
• at program execution time (only).
Type checking

- A strongly typed language guarantees that the compiler will accept only programs that execute without type errors, i.e., the compiler ensures that no type error will remain undetected.
- Strongly typed languages can use static and dynamic type checking.
- Standard example: array bounds checking.
  ```
  table: array[0...255] of char;
  i: integer
  table[i] cannot be guaranteed at compile time to fall in the range of 0 to 255
  ```
- In principle, type checking can be done entirely dynamically. However, this requires type tags for each value and each function application has to check for matching tags at run-time.
Type checking — discussion

compile–time:

+ points out type errors early

+ no overhead at program execution time (run–time)

− cannot always be done

? parts of the program that may never be executed are still checked (too restrictive)

program execution time:

+ allows more flexibility (e.g.: type may change depending on the use of a variable)

− overhead in terms of space and time

? part of program that are not executed are not checked

+ rapid prototyping

− harder to debug

Goal: Strongly typed languages with as much type checking as possible at compile–time
A simple type checker

Using a syntax-directed translation scheme (synthesized attributes only), we will describe a type checker for arrays, pointers, statements, and functions.

Grammar for source language:

\[
\begin{align*}
P & ::= D ; E \\
D & ::= D ; D \mid \text{id: } T \\
T & ::= \text{char} \mid \text{integer} \mid \text{array [num] of } T \mid \uparrow T \\
E & ::= \text{literal} \mid \text{num} \mid \text{id} \mid E \ mod \ E \mid E[E] \mid E \uparrow
\end{align*}
\]

- Basic types \textit{char, integer, typeError}
- assume all arrays start at 1, \textit{e.g.,}
  
  \text{array [256] of char}
  
  results in the type expression \textit{array}(1\ldots256,\textit{char})
- \uparrow builds a pointer type, so \uparrow \text{integer}
  
  results in the type expression \textit{pointer(}\textit{integer})
A simple type checker (cont.)

Partial translation scheme for the type system

\[
\begin{align*}
D & ::= \text{id}: T \quad \{ \text{addtype(id.entry, } T.\text{type}) \} \\
T & ::= \text{char} \quad \{ T.\text{type} \leftarrow \text{char} \} \\
T & ::= \text{integer} \quad \{ T.\text{type} \leftarrow \text{integer} \} \\
T & ::= \uparrow T_1 \quad \{ T.\text{type} \leftarrow \text{pointer}(T_1.\text{type}) \} \\
T & ::= \text{array [num]} \text{ of } T \quad \{ T.\text{type} \leftarrow \\
& \quad \text{array}(1 \ldots \text{num.val}, T_1.\text{type}) \}
\end{align*}
\]

These rules save the type of identifier \text{id} in the symbol table.
A simple type checker (cont.)

Type checking of expressions

E ::= literal  \{ E.type ← char \}
E ::= num  \{ E.type ← integer \}
E ::= id  \{ E.type ← lookup(id.entry) \}
E ::= E₁ mod E₂  \{ E.type ← if E₁.type = integer and E₂.type = integer then integer else typeError \}
E ::= E₁[E₂]  \{ E.type ← if E₂.type = integer and E₁.type = array(s,t) then t else typeError \}
E ::= E₁ ↑  \{ E.type ← if E₁.type = pointer(t) then t else typeError \}

Is our example language strongly typed?
Type checking statements

Statements do not typically have values, therefore we assign them the type *void*. If an error is detected within the statement, it gets type *typeError*.

\[
S ::= \text{id} \leftarrow E \quad \{ S.type \leftarrow \text{if id.type = E.type then void else typeError} \}
\]

\[
S ::= \text{if E then } S_1 \quad \{ S.type \leftarrow \text{if E.type = boolean then } S_1.type \text{ else typeError} \}
\]

\[
S ::= \text{while E do } S_1 \quad \{ S.type \leftarrow \text{if E.type = boolean then } S_1.type \text{ else typeError} \}
\]

\[
S ::= S_1 ; S_2 \quad \{ S.type \leftarrow \text{if } S_1.type = \text{void and } S_2.type = \text{void then void else typeError} \}
\]
Type checking functions

We add two new productions to the grammar to represent function declarations and applications.

\[
T ::= T \text{ "→" } T \\
E ::= E ( E )
\]

declaration

application

To capture the argument and return type, we use

\[
T ::= T_1 \text{ '→' } T_2 \left\{ \begin{array}{l}
T.type \leftarrow (T_1.type \rightarrow T_2.type) \\
E ::= E_1 ( E_2 ) \left\{ E.type \leftarrow \text{ if } E_2.type = s \text{ and } E_1.type = s \rightarrow t \text{ then } t \\
\text{ else } \text{typeError} \end{array} \right. \\
\end{array}
\]


Representation of type expressions

A convenient way to represent a type expression is to use a graph.

Example:

\[
\begin{array}{c}
\text{X} \\
\text{char} \\
\text{char}
\end{array}
\rightarrow
\begin{array}{c}
\text{integer} \\
\text{char}
\end{array}
\rightarrow
\begin{array}{c}
\text{X} \\
\text{char}
\end{array}
\rightarrow
\begin{array}{c}
\text{integer}
\end{array}
\]

\[(\text{char} \times \text{char}) \rightarrow \text{int}\]

This makes comparisons of type expressions easier, or helps implementing operations on type expressions such as “unification”.
Type names

Type expressions can be named and such names can occur in type expressions (e.g.: typedef in C).

When are two type expressions with type names the same?

*structural equivalence:*

- either same basic types (e.g., int is equivalent to int),
  or
- formed by application of the same type constructor to structurally equivalent types, where
  - type names are abbreviations for type expressions

In some sense, structural equivalence deletes names in composite constructs and replaces them by the type structures that they represent (unroll the tree or DAG).

*name equivalence:*

- Two type expressions are equivalent iff they are identical
  - each name is assumed to represent a different type.

\[
\text{type link}_1 = \uparrow A \neq \text{type link}_2 = \uparrow A
\]
Structural type equivalence algorithm

function sequiv (s,t): boolean;
begin
  if s and t are the same basic type then
  return true
  else if s = array (s₁, s₂) and t = array (t₁, t₂) then
  return sequiv (s₁, t₁) and sequiv (s₂, t₂)
  else if s = s₁ × s₂ and t = t₁ × t₂ then
  return sequiv (s₁, t₁) and sequiv (s₂, t₂)
  else if s = pointer (s₁) and t = pointer (t₁) then
  return sequiv (s₁, t₁)
  else if s = s₁ → s₂ and t = t₁ → t₂ then
  return sequiv (s₁, t₁) and sequiv (s₂, t₂)
  else
  return false
end

Problem: Recursive data types
Type equivalence

declaration equivalence

Problems arise in languages that use name equivalence but allow declarations to use a type that is not a name. Solution: Implicit new (fresh) type names are created for such types for each declaration.

type link = ↑ A
var \texttt{a}_1: ↑ A // implicit type name \texttt{type1}
var \texttt{a}_2, \texttt{a}_3: ↑ A // implicit type name \texttt{type2}
var \texttt{a}_4: \texttt{link}
var \texttt{a}_5: \texttt{link}

\texttt{a}_1 (\texttt{type1}) has a different type from \texttt{a}_2, \texttt{a}_3 (\texttt{type2}). \texttt{a}_4, \texttt{a}_5 are of the same type (\texttt{link}), but have a different type from \texttt{a}_1 and \texttt{a}_2, \texttt{a}_3.

Note: There are languages that use a hybrid approach.

Example: C uses structural equivalence for all types except structs (records). Structs may contain recursive references. C treats the name of the struct as part of its type, and therefore testing for structural equivalence stops when a struct (record) constructor is reached.
Type variables

Type expressions may contain variables (type variables) whose values are type expressions.

Type variables are used for implicitly typed languages or languages with polymorphic types.

Programming languages can be

- explicitly typed — every object is declared with its type (type checking)
- implicitly typed — type of object is derived from its use (type reconstruction)
- monomorphic — every object has a unique, single type
- polymorphic — allows objects to have more than one type (e.g., nil, and & in C)
Type variables — examples

Recall:

\[
\begin{array}{c}
E \vdash e_1: \text{integer} \\
E \vdash e_2: \text{integer}
\end{array}
\quad
\frac{}{E \vdash (e_1 + e_2): \text{integer}}
\]

where \(E\) is a type environment. In other words, “+” has the type expression \(\text{(integer × integer)} \to \text{integer}\).

What are the types of the variables \(a\) and \(b\) in the following program:

\[
\text{read}(a); \\
\text{read}(b); \\
\ldots \quad a + b \quad \ldots;
\]

Here is an idea: Guess the types of variables you don’t know about and use \textit{unification} to match guesses with rules.

\[
\begin{array}{l}
\text{read}(a) \quad \{a: \alpha\} \\
\text{read}(b) \quad \{a: \alpha, \ b: \beta\} \\
a + b \\
\quad \text{unify}(\alpha, \text{integer}) \\
\quad \text{unify}(\beta, \text{integer}) \\
\quad \text{apply type rule; result integer}
\end{array}
\]
Type variables — examples

- Polymorphic cons:

\[
E \vdash e_1 : \alpha \quad E \vdash e_2 : list(\alpha) \\
\frac{}{E \vdash \text{cons}(e_1, e_2) : list(\alpha)}
\]

cons has the type expression
\[
\forall \alpha. (\alpha \times list(\alpha)) \to list(\alpha)
\]

- Polymorphic nil:

\[
E \vdash \text{nil} : list(\alpha)
\]

nil has the type expression \(\forall \alpha. list(\alpha)\)

Questions:

- What is the type of \text{cons}(a,\text{nil})?
- What is the type of \text{cons}(1,\text{nil})?
- What is the type of \text{cons}(a,1)?
Unification

A unifier is a substitution of variables by expressions such that, when applied to two expressions $expr_1$ and $expr_2$, both expressions become syntactically identical. The most general unifier (mgu) makes as few and as general bindings as possible.

Example:

$\text{times}(Z, \text{times}(Y, 7) ) = \text{times}(4, W)$

$\Theta_1 = \{ Z \rightarrow 4, Y \rightarrow plus(3, 5), W \rightarrow times(plus(3, 5), 7) \}$

$\Theta_2 = \{ Z \rightarrow 4, W \rightarrow times(Y, 7) \}$

Question: Which of the two substitutions is more general?

Note: mgu $\theta$ is unique modulo renaming of variables
Type systems with polymorphism

How can we infer the type of a polymorphic function such as `length`?

```haskell
fun length (l) =
    if null(l) then 0
    else length(tl(l)) + 1;
```

- Need type variables to represent unknown types during type reconstruction (type inference).
- Need type environment $E$ that
  - Keeps track of type expressions for program variables and type variables
  - Is initialized to predefined types of constants and functions
Algorithm for typing functions

High-level view of algorithm

1. Add fresh type variables to $E$ for function name, arguments, and result. Add resulting type expression for function to $E$.

2. Analyze the program (parse tree) bottom up:

   • If constant, look-up type in $E$.
   • If variable, look-up type in $E$. If no entry in $E$, generate entry with fresh type variable.
   • If function application, look-up function in environment. "Apply" application inference rule by unification of type expressions of formal and actual parameters and result.

\[
E \vdash e_1 : \alpha \rightarrow \beta \quad E \vdash e_2 : \alpha \quad \overline{E \vdash e_1(e_2) : \beta}
\]

Note: If function is polymorphic ($\forall \alpha$. . . ) generate fresh type variable for bound variable and remove $\forall$ in resulting type expression.

If at any point unification fails, algorithm returns type error.
Algorithm for typing functions — example

Type inference for function \texttt{length}

Initial type environment:

\[ E = \{ 0:\text{integer}, \]
\[ 1:\text{integer}, \]
\[ \text{if} : \forall \alpha. \text{boolean} \times \alpha \times \alpha \rightarrow \alpha \]
\[ \text{null} : \forall \alpha. \text{list}(\alpha) \rightarrow \text{boolean} \]
\[ \text{tl} : \forall \alpha. \text{list}(\alpha) \rightarrow \text{list}(\alpha) \]
\[ + : \text{integer} \times \text{integer} \rightarrow \text{integer} \} \]

\begin{verbatim}
fun length (l) =
  if null(l) then 0
  else length(tl(l)) + 1;
\end{verbatim}

See ALSU 6.5.4, page 391 — 395
## Algorithm for typing functions — example

<table>
<thead>
<tr>
<th>Type expressions</th>
<th>Type environment</th>
</tr>
</thead>
<tbody>
<tr>
<td>l: ( \gamma )</td>
<td>l: ( \gamma )</td>
</tr>
<tr>
<td>length: ( \gamma \rightarrow \delta )</td>
<td>length: ( \gamma \rightarrow \delta )</td>
</tr>
<tr>
<td>null: ( list(\alpha_1) \rightarrow boolean )</td>
<td>( \gamma = list(\alpha_1) )</td>
</tr>
<tr>
<td>null(l): boolean</td>
<td></td>
</tr>
<tr>
<td>0: integer</td>
<td></td>
</tr>
<tr>
<td>l: ( list(\alpha_1) )</td>
<td>( \alpha_1 = \alpha_2 )</td>
</tr>
<tr>
<td>tl: ( list(\alpha_2) \rightarrow list(\alpha_2) )</td>
<td></td>
</tr>
<tr>
<td>tl(l): ( list(\alpha_1) )</td>
<td></td>
</tr>
<tr>
<td>length: ( list(\alpha_1) \rightarrow \delta )</td>
<td></td>
</tr>
<tr>
<td>length(tl(l)): ( \delta )</td>
<td></td>
</tr>
<tr>
<td>1: integer</td>
<td>( \delta = integer )</td>
</tr>
<tr>
<td>+: ( integer \times integer \rightarrow integer )</td>
<td></td>
</tr>
<tr>
<td>length(tl(l)) + 1: integer</td>
<td></td>
</tr>
<tr>
<td>if: ( boolean \times \alpha_3 \times \alpha_3 \rightarrow \alpha_3 )</td>
<td>( \alpha_3 = integer )</td>
</tr>
<tr>
<td>if(...): integer</td>
<td></td>
</tr>
<tr>
<td>length: ( list(\alpha_1) \rightarrow integer )</td>
<td></td>
</tr>
</tbody>
</table>

Since we don’t make any assumptions about \( \alpha_1 \), we get the polymorphic type for length:

\[ \forall \alpha_1. list(\alpha_1) \rightarrow integer. \]