Type system

• A type is a set of values and operations on those values
• A language’s type system specifies which operations are valid for a type
• The aim of type checking is to ensure that operations are used on the variable/expressions of the correct types
Type system ... 

• Languages can be divided into three categories with respect to the type:
  – “untyped”
    • No type checking needs to be done
    • Assembly languages
  – Statically typed
    • All type checking is done at compile time
    • Algol class of languages
    • Also, called strongly typed
  – Dynamically typed
    • Type checking is done at run time
    • Mostly functional languages like Lisp, Scheme etc.
Type systems ...

- **Static typing**
  - Catches most common programming errors at compile time
  - Avoids runtime overhead
  - May be restrictive in some situations
  - Rapid prototyping may be difficult

- Most code is written using static types languages

- In fact, developers for large/critical system insist that code be strongly type checked at compile time even if language is not strongly typed (use of Lint for C code, code compliance checkers)
Type System

• A type system is a collection of rules for assigning type expressions to various parts of a program
• Different type systems may be used by different compilers for the same language
• In Pascal type of an array includes the index set. Therefore, a function with an array parameter can only be applied to arrays with that index set
• Many Pascal compilers allow index set to be left unspecified when an array is passed as a parameter
Type system and type checking

- If both the operands of arithmetic operators +, -, \( x \) are integers then the result is of type integer.

- The result of unary & operator is a pointer to the object referred to by the operand.
  - If the type of operand is \( X \) the type of result is \textit{pointer to} \( X \).

- Basic types: integer, char, float, boolean.
- Sub range type: \( 1 \ldots 100 \).
- Enumerated type: (violet, indigo, red).
- Constructed type: array, record, pointers, functions.
Type expression

• Type of a language construct is denoted by a type expression
• It is either a basic type OR
• it is formed by applying operators called \textit{type constructor} to other type expressions

• A basic type is a type expression. There are two special basic types:
  – \textit{type error}: error during type checking
  – \textit{void}: no type value
• A type constructor applied to a type expression is a type expression
Type Constructors

- **Array**: if $T$ is a type expression then $\text{array}(I, T)$ is a type expression denoting the type of an array with elements of type $T$ and index set $I$

  ```
  int A[10];
  A can have type expression $\text{array}(0 .. 9, \text{integer})$
  ```
  - $C$ does not use this type, but uses equivalent of $\text{int}^*$

- **Product**: if $T_1$ and $T_2$ are type expressions then their Cartesian product $T_1 \times T_2$ is a type expression
  - $\text{Pair/tuple}$
Type constructors ...

- **Records**: it applies to a tuple formed from field names and field types. Consider the declaration
  
  ```
  type row = record
    addr : integer;
    lexeme : array [1 .. 15] of char
  end;
  
  var table: array [1 .. 10] of row;
  ```

- The type row has type expression
  
  ```
  record ((addr * integer) * (lexeme * array(1 .. 15, char)))
  ```

  and type expression of `table` is `array(1 .. 10, row)`
Type constructors ...

- **Pointer**: if $T$ is a type expression then $\text{pointer}(T)$ is a type expression denoting type pointer to an object of type $T$

- **Function**: function maps domain set to range set. It is denoted by type expression $D \to R$
  - For example `%` has type expression $\text{int} \to \text{int}$
  - The type of function $\text{int}\star f(\text{char} \ a, \ \text{char} \ b)$ is denoted by $\text{char} \ \star \ \text{char} \ \to \ \text{pointer}(\text{int})$
Specifications of a type checker

• Consider a language which consists of a sequence of declarations followed by a single expression

P → D ; E
D → D ; D | id : T
T → char | integer | T[num] | T*
E → literal | num | E%E | E [E] | *E
Specifications of a type checker ...

• A program generated by this grammar is

```plaintext
key : integer;
key %1999
```

• Assume following:
  – basic types are char, int, type-error
  – all arrays start at 0
  – char[256] has type expression array(0 .. 255, char)
## Rules for Symbol Table entry

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D → id : T</td>
<td><code>addtype(id.entry, T.type)</code></td>
</tr>
<tr>
<td>T → char</td>
<td>T.type = char</td>
</tr>
<tr>
<td>T → integer</td>
<td>T.type = int</td>
</tr>
<tr>
<td>T → T₁*</td>
<td>T.type = pointer(T₁.type)</td>
</tr>
<tr>
<td>T → T₁[num]</td>
<td>T.type = array(0..num-1, T₁.type)</td>
</tr>
</tbody>
</table>
Type checking of functions

E → E1 (E2)  E. type =
    (E1.type == s → t and  E2.type == s)
? t : type-error
Type checking for expressions

E → literal
E → num
E → id
E → E₁ % E₂

E → E₁[E₂]

E → *E₁
Type checking for expressions

\begin{align*}
E \to \text{literal} \quad & E.\text{type} = \text{char} \\
E \to \text{num} \quad & E.\text{type} = \text{integer} \\
E \to \text{id} \quad & E.\text{type} = \text{lookup(id.entry)} \\
E \to E_1 \% E_2 \quad & E.\text{type} = \begin{cases} 
\text{integer} & \text{if } E_1.\text{type} == \text{integer} \text{ and } E_2.\text{type} == \text{integer} \\
\text{type_error} & \text{otherwise}
\end{cases} \\
E \to E_1[E_2] \quad & E.\text{type} = \begin{cases} 
\text{t} & \text{if } E_2.\text{type} == \text{integer} \text{ and } E_1.\text{type} == \text{array(s,t)} \\
\text{type_error} & \text{otherwise}
\end{cases} \\
E \to *E_1 \quad & E.\text{type} = \begin{cases} 
\text{t} & \text{if } E_1.\text{type} == \text{pointer(t)} \\
\text{type_error} & \text{otherwise}
\end{cases}
\end{align*}
Type checking for statements

• Statements typically do not have values. Special basic type *void* can be assigned to them.

\[
S \rightarrow \text{id := } E \\
S \rightarrow \text{if } E \text{ then } S1 \\
S \rightarrow \text{while } E \text{ do } S1 \\
S \rightarrow S1 ; S2
\]
Type checking for statements

- Statements typically do not have values. Special basic type `void` can be assigned to them.

\[
\begin{align*}
S & \rightarrow \text{id := E} & S.\text{Type} &= \text{if id.type == E.type then void else type_error} \\
S & \rightarrow \text{if E then S1} & S.\text{Type} &= \text{if E.type == boolean then S1.type else type_error} \\
S & \rightarrow \text{while E do S1} & S.\text{Type} &= \text{if E.type == boolean then S1.type else type_error} \\
S & \rightarrow S1 ; S2 & S.\text{Type} &= \text{if S1.type == void and S2.type == void then void else type_error}
\end{align*}
\]
Equivalence of Type expression

• Structural equivalence: Two type expressions are equivalent if
  • either these are same basic types
  • or these are formed by applying same constructor to equivalent types

• Name equivalence: types can be given names
  • Two type expressions are equivalent if they have the same name
Function to test structural equivalence

boolean sequiv(type s, type t) :
   If s and t are same basic types
   then return true
   elseif s == array(s1, s2) and t == array(t1, t2)
       then return sequiv(s1, t1) && sequiv(s2, t2)
   elseif s == s1 \* s2 and t == t1 \* t2
       then return sequiv(s1, t1) && sequiv(s2, t2)
   elseif s == pointer(s1) and t == pointer(t1)
       then return sequiv(s1, t1)
   elseif s == s1 \rightarrow s2 and t == t1 \rightarrow t2
       then return sequiv(s1,t1) && sequiv(s2,t2)
   else return false;
Efficient implementation

- Bit vectors can be used to represent type expressions. Refer to: A Tour Through the Portable C Compiler: S. C. Johnson, 1979.

<table>
<thead>
<tr>
<th>Basic type</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>0000</td>
</tr>
<tr>
<td>Char</td>
<td>0001</td>
</tr>
<tr>
<td>Integer</td>
<td>0010</td>
</tr>
<tr>
<td>real</td>
<td>0011</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type constructor</th>
<th>encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>pointer</td>
<td>01</td>
</tr>
<tr>
<td>array</td>
<td>10</td>
</tr>
<tr>
<td>function</td>
<td>11</td>
</tr>
</tbody>
</table>
Efficient implementation ...

<table>
<thead>
<tr>
<th>Basic type</th>
<th>Encoding</th>
<th>Type constructor</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>0000</td>
<td>pointer</td>
<td>01</td>
</tr>
<tr>
<td>Char</td>
<td>0001</td>
<td>array</td>
<td>10</td>
</tr>
<tr>
<td>Integer</td>
<td>0010</td>
<td>function</td>
<td>11</td>
</tr>
<tr>
<td>real</td>
<td>0011</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Type expression encoding

char 000000 0001
function( char ) 000011 0001
pointer( function( char ) ) 000111 0001
array( pointer( function( char) ) ) 100111 0001

This representation saves space and keeps track of constructors
Checking name equivalence

- Consider following declarations
  ```
  typedef cell* link;
  link next, last;
  cell  *p, *q, *r;
  ```
- Do the variables next, last, p, q and r have identical types?
- Type expressions have names and names appear in type expressions.
- Name equivalence views each type name as a distinct type
Name equivalence ...

<table>
<thead>
<tr>
<th>variable</th>
<th>type expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>next</td>
<td>link</td>
</tr>
<tr>
<td>last</td>
<td>link</td>
</tr>
<tr>
<td>p</td>
<td>pointer(cell)</td>
</tr>
<tr>
<td>q</td>
<td>pointer(cell)</td>
</tr>
<tr>
<td>r</td>
<td>pointer(cell)</td>
</tr>
</tbody>
</table>

- Under name equivalence next = last and p = q = r, however, next ≠ p
- Under structural equivalence all the variables are of the same type
Some compilers allow type expressions to have names. However, some compilers assign implicit type names. A fresh implicit name is created every time a type name appears in declarations. Consider:

```plaintext
type link = ^ cell;
var next : link;
    last : link;
    p, q : ^ cell;
    r : ^ cell;
```

In this case type expression of q and r are given different implicit names and therefore, those are not of the same type.
Name equivalence ...

The previous code is equivalent to

```plaintext
type link = ^ cell;
    np = ^ cell;
    nr = ^ cell;
var next : link;
    last : link;
    p, q: np;
    r : nr;
```
Cycles in representation of types

- Data structures like linked lists are defined recursively
- Implemented through structures which contain pointers to structure
- Consider following code

```pascal
type link = ^ cell;
cell = record
    info : integer;
    next : link
end;
```

- The type name `cell` is defined in terms of `link` and `link` is defined in terms of `cell` (recursive definitions)
Cycles in representation of ...

- Recursively defined type names can be substituted by definitions.
- However, it introduces cycles into the type graph.

```pascal
link = ^ cell;
cell = record
  info : integer;
  next : link
end;
```
Cycles in representation of ...

• C uses structural equivalence for all types except records (struct)
• It uses the acyclic structure of the type graph
• Type names must be declared before they are used
  – However, allow pointers to undeclared record types
  – All potential cycles are due to pointers to records
• Name of a record is part of its type
  – Testing for structural equivalence stops when a record constructor is reached
Type conversion

• Consider expression like $x + i$ where $x$ is of type real and $i$ is of type integer
• Internal representations of integers and reals are different in a computer
  – different machine instructions are used for operations on integers and reals
• The compiler has to convert both the operands to the same type
• Language definition specifies what conversions are necessary.
Type conversion ...

- Usually conversion is to the type of the left hand side
- Type checker is used to insert conversion operations:
  \[ x + i \rightarrow x \text{ real+ inttoreal}(i) \]
- Type conversion is called implicit/coercion if done by compiler.
- It is limited to the situations where no information is lost
- Conversions are explicit if programmer has to write something to cause conversion
Type checking for expressions

\[ E \rightarrow \text{num} \quad \Rightarrow \quad \text{E.type} = \text{int} \]
\[ E \rightarrow \text{num.num} \quad \Rightarrow \quad \text{E.type} = \text{real} \]
\[ E \rightarrow \text{id} \quad \Rightarrow \quad \text{E.type} = \text{lookup( id.entry )} \]
\[ E \rightarrow E_1 \text{ op } E_2 \quad \Rightarrow \quad \text{E.type} = \]
\[ \quad \text{if } E_1.\text{type} == \text{int} \quad \&\& \quad E_2.\text{type} == \text{int} \]
\[ \quad \text{then } \text{int} \]
\[ \quad \text{elif } E_1.\text{type} == \text{int} \quad \&\& \quad E_2.\text{type} == \text{real} \]
\[ \quad \text{then } \text{real} \]
\[ \quad \text{elif } E_1.\text{type} == \text{real} \quad \&\& \quad E_2.\text{type} == \text{int} \]
\[ \quad \text{then } \text{real} \]
\[ \quad \text{elif } E_1.\text{type} == \text{real} \quad \&\& \quad E_2.\text{type} == \text{real} \]
\[ \quad \text{then } \text{real} \]
Overloaded functions and operators

• Overloaded symbol has different meaning depending upon the context
• In math, + is overloaded; used for integer, real, complex, matrices
• In Ada, () is overloaded; used for array, function call, type conversion
• Overloading is resolved when a unique meaning for an occurrence of a symbol is determined
Overloaded functions and operators

• In Ada standard interpretation of * is multiplication of integers
• However, it may be overloaded by saying
  function “*” (i, j: integer) return complex;
  function “*” (i, j: complex) return complex;
• Possible type expression for “*” include:
  integer x integer → integer
  integer x integer → complex
  complex x complex → complex
Overloaded function resolution

• Suppose only possible type for \(2, 3\) and \(5\) is integer

• \(Z\) is a complex variable

• \(3*5\) is either integer or complex depending upon the context
  – in \(2*(3*5)\): \(3*5\) is integer because \(2\) is integer
  – in \(Z*(3*5)\): \(3*5\) is complex because \(Z\) is complex
Type resolution

• Try all possible types of each overloaded function (possible but brute force method!)
• Keep track of all possible types
• Discard invalid possibilities
• At the end, check if there is a single unique type
• Overloading can be resolved in two passes:
  – Bottom up: compute set of all possible types for each expression
  – Top down: narrow set of possible types based on what could be used in an expression
Determining set of possible types

\[ E' \rightarrow E \quad E'.\text{types} = E.\text{types} \]

\[ E \rightarrow \text{id} \quad E.\text{types} = \text{lookup(id)} \]

\[ E \rightarrow E_1(E_2) \quad E.\text{types} = \]
\[ \{ t \mid \exists s \in E_2.\text{types} \land s \rightarrow t \text{ is in } E_1.\text{types} \} \]

```
{ixi \rightarrow i
ixi \rightarrow c
cxc \rightarrow c}
```
Narrowing the set of possible types

• Ada requires a complete expression to have a unique type
• Given a unique type from the context we can narrow down the type choices for each expression
• If this process does not result in a unique type for each sub expression then a type error is declared for the expression
Narrowing the set of ...

$E' \rightarrow E$ \hspace{1cm} $E'.\text{types} = E.\text{types}$

$E \rightarrow \text{id}$ \hspace{1cm} $E.\text{types} = \text{lookup(id)}$

$E \rightarrow E_1(E_2)$ \hspace{1cm} $E.\text{types} =$

\[
\{ t \mid \exists s \in E_2.\text{types} \land s \rightarrow t \text{ is in } E_1.\text{types} \}
\]
E' → E   E'.types = E.types
      E.unique = if E'.types == {t} then t
                  else type_error
E → id    E.types = lookup(id)
E → E_1(E_2)  E.types =
              \{ t | ∃ s in E_2.types & & s → t is in E_1.types \}
              t = E.unique
S = \{ s | s ∈ E_2.types and (s → t) ∈ E_1.types \}
E_2.unique = if S == {s} then s else type_error
E_1.unique = if S == {s} then s → t
              else type_error
Polymorphic functions

• A function can be invoked with arguments of different types

• Built in operators for indexing arrays, applying functions, and manipulating pointers are usually polymorphic

• Extend type expressions to include expressions with type variables

• Facilitate the implementation of algorithms that manipulate data structures (regardless of types of elements)
  – Determine length of the list without knowing types of the elements
Polymorphic functions ...

- Strongly typed languages can make programming very tedious
- Consider identity function written in a language like Pascal
  \begin{verbatim}
  function identity (x: integer): integer;
  \end{verbatim}
  - This function is the identity on integers: \texttt{int} \(\rightarrow\) \texttt{int}
  - If we want to write identity function on char then we must write
    \begin{verbatim}
    function identity (x: char): char;
    \end{verbatim}
  - This is the same code; only types have changed. However, in Pascal a new identity function must be written for each type
  - Templates solve this problem somewhat, for end-users
    \begin{itemize}
    \item For compiler, multiple definitions still present!
    \end{itemize}
Type variables

• Variables can be used in type expressions to represent unknown types
• **Important use:** check consistent use of an identifier in a language that does not require identifiers to be declared
• An inconsistent use is reported as an error
• If the variable is always used as of the same type then the use is consistent and has lead to type inference
• Type inference: determine the type of a variable/language construct from the way it is used
  – Infer type of a function from its body
function deref(p) { return *p; }

• Initially, nothing is known about type of p
  – Represent it by a type variable
• Operator * takes pointer to an object and returns the object
• Therefore, p must be pointer to an object of unknown type α
  – If type of p is represented by β then
    β=pointer(α)
  – Expression *p has type α
• Type expression for function deref is
  for any type α:  pointer(α) → α
• For identity function, the type expression is
  for any type α:  α → α
Reading assignment

- Rest of Section 6.6 and Section 6.7 of Old Dragonbook [Aho, Sethi and Ullman]