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 ***pointer to X***
- **Basic types:** integer, char, float, boolean
- **Sub range type**: 1 … 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 *type constructor* to other type expressions

- A basic type is a type expression. There are two special basic types:
  - *type error*: error during type checking
  - *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 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 array(0 .. 9, integer)

  - C does not use this type, but uses equivalent of int*

- Product: if T1 and T2 are type expressions then their Cartesian product T1 * T2 is a type expression
  - 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 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 → R
  - For example % has type expression int * int → int
  - The type of function int* f(char a, char b) is denoted by char * char → pointer(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

  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

D → id : T	addtype(id.entry, T.type)
T → char	T.type = char
T → integer	T.type = int
T → T_1*	T.type = pointer(T_1.type)
T → T_1 [num]	T.type = array(0..num-1, T_1.type)

Type checking of functions

E → E1 (E2)   E. type =

(E1.type == s → t and E2.type == s)

? t : type-error

Type checking for expressions

$E \rightarrow$ literal

$E \rightarrow$ num

$E \rightarrow$ id

$E \rightarrow E_1 \% E_2$

$E \rightarrow E_1[E_2]$

$E \rightarrow *E_1$

Type checking for expressions

$E \rightarrow$ literal	E.type = char
$E \rightarrow$ num	E.type = integer
$E \rightarrow$ id	E.type = lookup(id.entry)
$E \rightarrow E_1 \% E_2$	E.type = if E_1.type == integer and E_2.type==integer then integer else type_error
$E \rightarrow E_1[E_2]$	E.type = if E_2.type==integer and E_1.type==array(s,t) then t else type_error
$E \rightarrow *E_1$	E.type = if E_1.type==pointer(t) then t else type_error

Type checking for statements

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

  S → id := E

  S → if E then S1

  S → while E do S1

  S → S1 ; S2

Type checking for statements

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

S → id := E	S.Type = if id.type == E.type then void else type_error
S → if E then S1	S.Type = if E.type == boolean then S1.type else type_error
S → while E do S1	S.Type = if E.type == boolean then S1.type else type_error
S → S1 ; S2	S.Type = if S1.type == void and S2.type == void then void else type_error

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→s2  and  t == t1→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.

Basic type	Encoding
Boolean	0000
Char	0001
Integer	0010
real	0011

Type constructor	encoding
pointer	01
array	10
function	11

Efficient implementation …

Basic type	Encoding
Boolean	0000
Char	0001
Integer	0010
real	0011

Type constructor	Encoding
pointer	01
array	10
function	11

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 …

variable	type expression
next	link
last	link
p	pointer(cell)
q	pointer(cell)
r	pointer(cell)

- Under name equivalence next = last and p = q = r , however, next ≠ p
- Under structural equivalence all the variables are of the same type

Name equivalence …

- 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

```
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

```
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

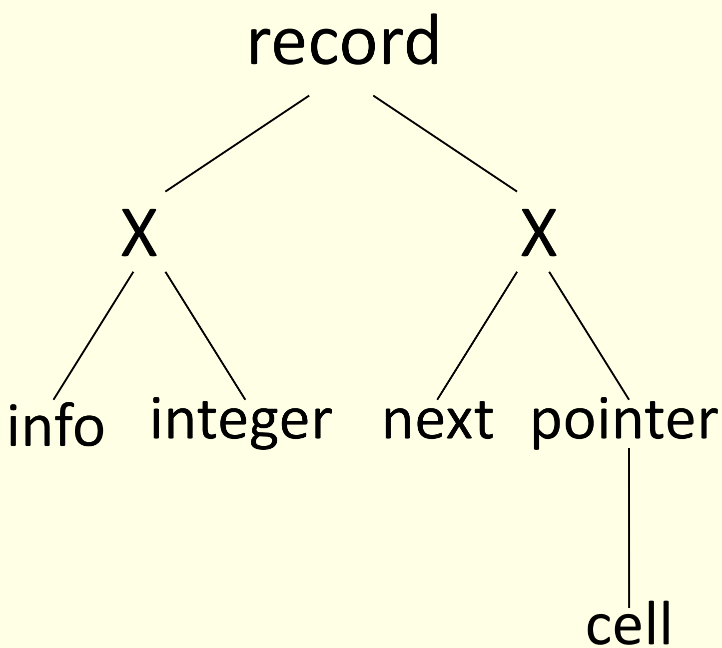
```
type link = ^ cell;
     cell = record
                info : integer;
                next : link
            end;
```

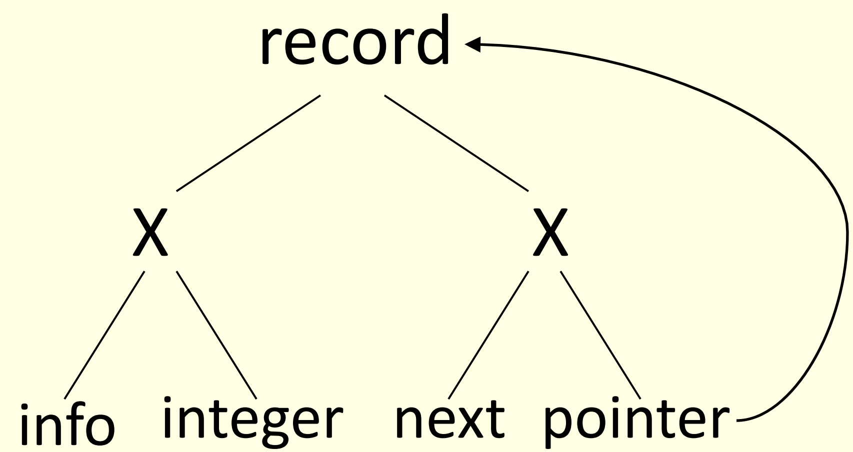
- The type name cell is defined in terms of link and link is defined in terms of cell (recursive definitions)

Cycles in representation of …

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

```
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

  ⇨ x 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

$E \rightarrow$ num	E.type = int
$E \rightarrow$ num.num	E.type = real
$E \rightarrow$ id	E.type = lookup(id.entry)

$E \rightarrow E_1$ op E_2

E.type =

if E_1.type == int && E_2.type == int
then int
elif E_1.type == int && E_2.type == real
then real
elif E_1.type == real && E_2.type == int
then real
elif E_1.type == real && E_2.type==real
then 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

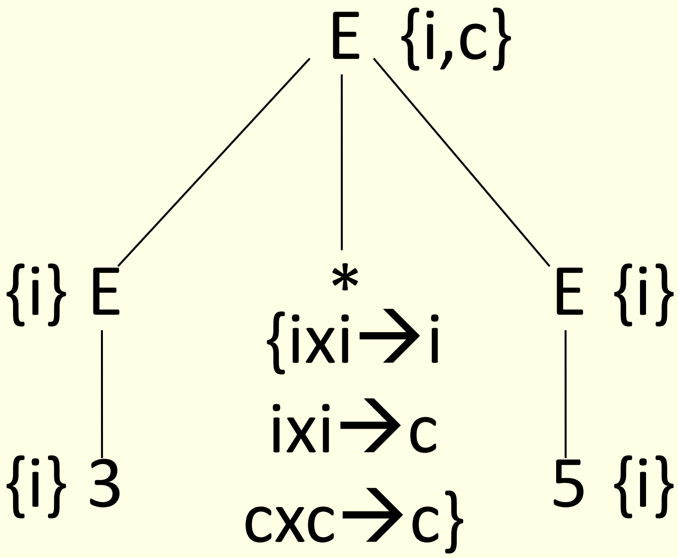
Determining set of possible types

$E' \rightarrow E$ $\quad E'.types = E.types$

$E \rightarrow id$ $\quad E.types = lookup(id)$

$E \rightarrow E_1(E_2)$ $\quad E.types =$

$\{t \mid \exists s \text{ in } E_2.types \ \&\& \ s \rightarrow t \text{ is in } E_1.types\}$



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’ → E $\quad$ E’.types = E.types

E → id $\quad$ E.types = lookup(id)

E → $E_1(E_2)$ $\quad$ E.types =

{ t | ∃s in E_2.types && s→t is in E_1.types}

Narrowing the set of …

$E' \rightarrow E$	$E'.types = E.types$
	$E.unique$ = if $E'.types==\{t\}$ then t else type_error
$E \rightarrow id$	$E.types = lookup(id)$
$E \rightarrow E_1(E_2)$	$E.types = \{ t \mid \exists s \text{ in } E_2.types \;\&\&\; s \rightarrow t \text{ is in } E_1.types\}$
	$t = E.unique$
	$S = \{s \mid s \in E2.types \text{ and } (s \rightarrow t) \in E1.types\}$
	$E_2.unique$ = if $S==\{s\}$ then s else type_error
	$E_1.unique$ = if $S==\{s\}$ then $s \rightarrow t$ 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
  function identity (x: integer): integer;
- This function is the identity on integers: int → int
- If we want to write identity function on char then we must write
  function identity (x: char): char;
- 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
  - For compiler, multiple definitions still present!

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 $\beta = pointer(\alpha)$
  - Expression *p has type α
- Type expression for function deref is
  for any type α: $pointer(\alpha) \rightarrow \alpha$
- For identity function, the type expression is
  for any type α: $\alpha \rightarrow \alpha$

Reading assignment

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