Types

- What is a type?
- Type checking
- Type conversion
- *Aggregates: arrays*

What is a type?

- *A set of values and the valid operations on those values*
  - Integers + - * *div* < <= = >= > ...
  - Arrays:
    `lookUp(<array>,<index>)`
    `assign(<array>,<index>,<value>)`
    `initialize(<array>), setBounds(<array>)`
  - User-defined types:
    `Java interfaces`
- Program semantics (meaning) embedded in types used
  - *Additional correctness check provided beyond valid syntax*

3 Views of Types

- **Set point of view:**
  - *int* = `{1,-2,... }`
  - *char* = `{'a','b',...}`
  - *list* = `{ (), (a (2 b) ), ... }`
- **Abstraction point of view:**
  - Set of operations which can be combined meaningfully

    e.g., *Java interfaces*

3 Views of Types

- **Constructive point of view**
  - **Primitive types** e.g., *int, char, bool, enum*{red,green,yellow}
  - **Composite/constructed types:**
    - reference e.g., *pointerTo*(**int**)
    - array e.g., *arrayOf*(**char**) or *arrayOf*(**char, 20**) or ...
    - record/structure e.g., *record*(**age:int, name:string**)
    - union e.g. *union*(**int, pointerTo(char)**)
    - list e.g., *list*(...)
    - function e.g., *float* → *int*

    CAN BE NESTED! pointerTo(arrayOf(pointerTo(char)))

Types

- Implicit
  - If variables are typed by usage
    - Prolog, Scheme, Lisp, Smalltalk
- Explicit
  - If declarations bind types to variables at compile time
    - Pascal, Algol68, C, C++, Java
- Mixture
  - Implicit by default but allows explicit declarations
    - Haskell, ML

Type System

- Rules for constructing types
- Rules for determining/inferring the type of expressions
- Rules for type compatibility:
  - In what contexts can values of a type be used (e.g., in assignment, as arguments of functions,...)
- Rules for type equivalence or type conversion
  - Determining (ensuring) that an expression can be used in some context

Types of Expressions

- If *f* has type $S \rightarrow T$ and *x* has type *S*, then *f(x)* has type *T*
  - type of *`3 div 2`* is *int*
  - type of *`round(3.5)`* is *int*
- *Type error* - using wrongly typed operands in an operation
  - `round("Nancy")`
  - `3.5 div 2`
  - `"abc"+ 3`

Type Checking

- *Goal:* to find out as early as possible, if each procedure and operator is supplied with the correct type of arguments
  - Type error: when a type is used improperly in a context
  - Type checking performed to prevent type errors
- Modern PLs often designed to do type checking (as much as possible) during compilation

Type Checking

- *Compile-time* (static)
  - At compile time, uses declaration information or can infer types from variable uses
- *Run-time* (dynamic)
  - During execution, checks type of object before doing operations on it
    - Uses type tags to record types of variables
- Combined (compile- and run-time)

Type Safety

- A *type safe* program executes on all inputs without type errors
  - Goal of type checking is to ensure type safety
  - Type safe does not mean without errors

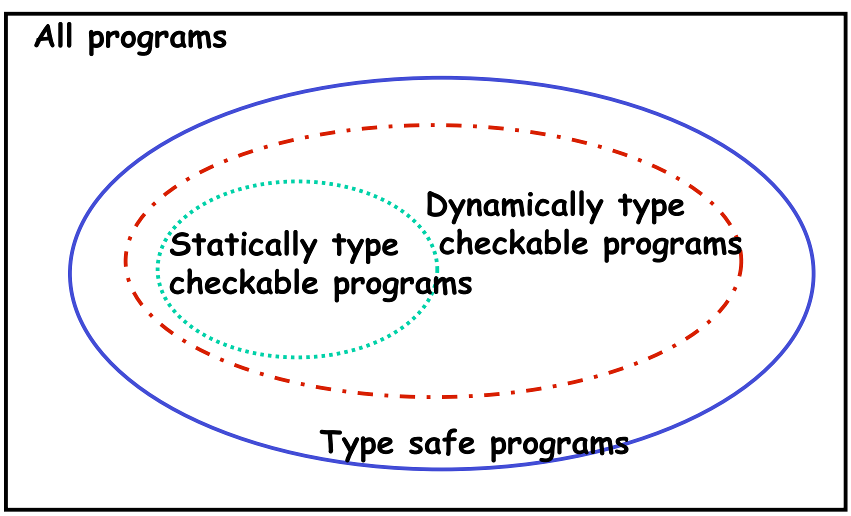
```
read n;
if n>0 then {y:="ab";
            if n<0 then x := y-5;}
```

    - Note that assignment to x is never executed so program is *type safe* (but contains an error).

Strong Typing

- *Strongly typed PL* By definition, PL requires all programs to be type checkable
- *Statically strongly typed PL* - compiler allows only programs that can be type checked fully at compile time
  - Algol68, ML
- *Dynamically strongly typed PL* -Operations include code to check run-time types of operands, if type cannot be determined at compile time
  - Pascal, Java

Hierarchy of Programs



Type Checking

- Kind of types used is orthogonal to when complete type checking can be accomplished.

	static checking	dynamic checking
Implicit types	ML	Scheme
Explicit types	Algol68	C, Pascal

Difficulties in Static Type Checking

- If validity of expression depends not only on the types of the operands but on their values, static type checking cannot be accomplished
  - Taking successors of enumeration types
  - Using unions without type test guard
  - Converting ranges into subranges
  - Reading values from input
  - Dereferencing void * pointers

Type Conversion

- Implicit conversion - *coercion*
  - In C, mixed mode numerical operations
    - `double d,e;…e=d+2;` //2 coerced to 2.0
  - Usually can use *widening* or conversion without loss of precision
    - integer → double, float → double
    - But real → int may lose precision and therefore cannot be implicitly coerced!
  - Cannot coerce user-defined types or structures

Types Require Work

- for programmer - has to start typing process
  - Usually needs declarations for user-defined constants, variables, functions
    - *e.g.* procedural languages: C,C++,Pascal, Ada,...
- for PL implementer
  - Implementing type checking
  - For dynamically typed languages, carrying around type information with (all/some) values at runtime -- wastes space and time
- for PL designer
  - Balance tradeoffs above.

Type Conversion

- Explicit conversion
  - In Pascal, can explicitly convert types which may lose precision (*narrowing*)
    - `round(s)` real → int by rounding
    - `trunc(s)` real → int by truncating
  - In C, casting sometimes is explicit conversion
    - `dqstr((double) n)` where `n` is declared to be an `int`
    - `freelist *s; ... (char *)s;` forces `s` to be considered as pointing to a char for purposes of pointer arithmetic

Overloading Operators

- Primitive type of *polymorphism*
  - When an operator allows operands of more than one type, in different contexts
- Examples
  - Addition: 2+3 is 5, versus concatenation: "abc"+"def" is "abcdef"
  - Comparison operator used for two different types: 2 == 3 versus "abc" == "def"
  - Integer addition: 1+2 versus real addition: 1.+2.

Primitive Types

- Issues
  - type checking
  - representation in the machine
- Boolean
  - use of integer 0/non-0 versus `true/false`
- Char versus string
- Integer
  - length fixed by standards or implementation (portability issues)
  - multiple lengths (C: short, int, long)
  - signs
- Float/real (all issues of ints plus)
  - should value comparison be allowed?
  - rep: sign(1 bit)/mantissa(23 bits)/exponent(8 bits)

Definition of Arrays

- Homogeneous, indexed collection of values
- Access to individual elements through subscript
- Choices made by a PL designer
  - Subscript syntax
  - Subscript type, element type
  - When to set bounds, compile-time or runtime?
  - How to initialize?
  - What built-in operations allowed?

Array Type

- What is part of the array type?
  - Size?
  - Bounds?
    - Pascal: bounds are part of type
    - C,Algol68: bounds are not part of type
    - Must be fixed at compile-time in Pascal but can be set at runtime in C and Fortran
  - Dimension? always part of the type
- Choice has ramifications on kind of type checking needed

Choices for Arrays

- Global lifetime, static shape (in global memory)
- Local lifetime
  - Static shape (kept in fixed length portion of frame)
  - Shape bound at elaboration time when control enters a scope
    - (e.g., Ada, Fortran allow defn of array bounds when fcn is elaborated; kept in variable length portion of frame)
- Arrays as objects (Java)
  - Shape bound at elaboration time (kept in heap)
    - `int[ ] a;…a = new int[size]`
- Dynamic shape (can change during execution) must be kept on heap

Arrays in Algol68

- Array type only includes dimensionality, not bounds

  `[1:12] int month;[1:7] int day;` *row int*

  `[0:10,0:10]real matrix;`

  `[-4:10,6:9]real table` *row row real*

  Note `table` and `matrix` are type equivalent!

- Example - [1:10] [1:5,1:5] int kinglear;

  kinglear is a vector of 10 elements each of which is a *row row int* array of 25 elements, so kinglear is of type *row of (row row int)* in contrast to the type *row row row int*

  kinglear[j] is legal wherever *row row int* is legal

  kinglear[j][1,2] is legal wherever *int* is legal

  kinglear[1, 2, 3] is ILLEGAL!

Algol 68 Array Operations

- *Trimming:* yields some cross section of an original Algol68 array (slicing an array into subarrays)
- *Subscripting:* limiting 1 dimension to a single index value

  `[1:10]int a,b; [1:20]real x; [1:20,1:20]real xx;`

  `b[1:4] := a[1:4]` -- assigns 4 elements

  `b:= a` -- assigns all of a to b, same effect as `b[1:10]:=a[1:10]`

  `xx[4,1:20]:= x` --assigns 20 elements to row 4 of xx

  `xx[8:9,7] := x[1:2]` --assigns `x[1]` to `xx[8,7]` and `x[2]` to `xx[9,7]`

Arrays -Implementation

- For fixed length array, symbol table keeps track of name, element type, bounds etc. during compilation; can allocate in static storage or on frame of declaring method.
- For arrays whose length is not knowable at compile-time, we use a *dope vector,* a descriptor of fixed size on the stack frame, and then allocate space for the array data separately
- Dope vector contains:
  - Name, type of subscript, bounds, type of elements, number of bytes in each element, pointer to first storage location of array
  - Allows calculation of actual frame address of an array element from these values

Array Addressing

- X[low:high] of E bytes each data item. What's the address of X[j]?

  addr(X) + (j-low) *E <= addr(X) +(high-low)*E
  - Note: addr(X)-low*E is a compile-time constant
  - X[ ] row real (4 bytes each);
  - X[3] is addr(X[0]) + (3-0)*4 = addr(X) + 12
  - X[0], X[1] is at address X[0]+4, X[2] is at address X[0]+8, etc

Array Addressing

- Assume arrays are stored in row major order y[0,0], y[0,1], y[0,2], …, y[1,*], y[2,*],…
- Consider memory a sequence of locations
- Then if have y[low1:hi1,low2:hi2] in Algol68, location y[j,k] is

addr(y[low1,low2]) + (hi2-low2+1)*E*(j-low1)+(k-low2)*E

#locs per row *#rows in front of row j* *# elements in row j in front of element [j,k]*

Example

y[0:2, 0:5] in Algol68, an int array. Assume row major storage and find address of y[1,3].

address of y[1,3] = addr(y[0,0])+(5-0+1)*4*(1-0)+(3-0)*4

6 elements per row

1 row before row 1

3 elements in row 1 before 3

= addr(y[0,0])+24+12

= addr(y[0,0])+36

- Analogous formula holds for column major order.