

# Type Checking: Varieties of Programs

All Programs:

```
a="bA1D" + 4.0;
```

**Type-safe programs : execute without type error**

```
Int n; PrimeInteger k; //set of primes
read(n); k=n; /* determining if a number is
prime needs to be done at run time; & is*very*
expensive */
Formula f; //set of all predicate calc. formulas
AlwaysTrueFormula h; //always true: p v ~p
thm=f; /* no algorithm is even possible
for determining if a formula is a theorem! */
```

**Programs in strongly-typed languages must, by definition, execute type-safely. (Obligation on compilers)**

**Dynamically Strongly-typed Language -- uses (a lot of) run time checks, and carries around type with values.**

```
if (n>0) {x=3;};
else if (n>0) x='a'+2; //never executed
```

**Statically Strongly-typed Language**

## Type equivalence rules

**SE1.** Type *name* N is equiv to itself

**SE2.** If T1 is equiv to T2, and k is a type constructor (e.g., k is 'pointerTo', 'array') then k(T1) is equiv to k(T2)

**SE3.** If you have type definition `typedef T id;` then `id` is equiv to T

**SE3'.** If you have type definition which is just an alias `typedef Id1 Id2;` then `Id1` and `Id2` are equiv

### Some reasonable combinations

- SE1 & SE2 = "strict name equivalence"; has desirable effect of keeping `USstate` and `Element` and `char[2]` from mixing, when

*type USstate=char[2]; type Element=char[2];chemical*

- SE1 & SE2 & SE3 = "structural type equivalence"; 'same shape' values; easy to implement and remember; but does not distinguish the above 2 types.
- SE3' instead of SE3: "weak name equivalence"; Why? recall that in defining `Stack` we used `StackElement` to describe type of values on the stack (rather than `int`) so we could easily change the kind of stack we have just by changing `typedef int StkElt;` to `typedef *char StkElt;` (a kind of type parameter). Strict name equivalence says this is not OK: for a stack of integers we cannot do `int w; push(w)` since `push` required `StkElt` value as argument, yet gets an `int`. SE3' allows for this.
- Best of both worlds: in Ada, SE1 & SE2 & programmer's choice:

`type S is new T;` // S and T are not equivalent;

`type S derived from T;` // S & T are equivalent=SE3

## Some possible situations

Consider the following type and variable declarations:

```
type A = pointerTo int;
  var a,b : A;
type D = pointerTo int;
  var c,d : D;
type StackElementType = A;
  var s : StackElementType;
var f,g : pointerTo int;
var h : pointerTo int;
```

For each of the following statements and each kind of type equivalence on the previous slide, determine whether the assignment is type correct or not:

```
a=b;
a=c;
a=f;
s=a;
f=g;
f=h;
```

/\* There is a weird implementation of Pascal where even **f=h;** leads to a type error! The only variables of equivalent composite type are those declared in the **same** declaration statement!?!?!

Happened when the language specification did not explicitly describe rules of type equivalence, and implementors chose whatever was easiest.

We have learned since then (we hope)

# PL Types: more details

For each kind of type, look at:

- variability between languages
- type checking issues
- implementation/mapping to hardware

## **BASIC TYPES**

### 1. BOOLEAN

C and Lisp/Scheme use integer 0/non-0 instead; bad for type checking (in C even adding `enum Bool {true, false}` won't help because of structural equivalence. So why do it? Hardware supports instructions 'branch on zero/non-zero'. When functions return special values for fail, if you use 0, you can save a test: `if (find(a)) { }` instead of `if (find(a) == 0)`

### 2. CHARACTER

Languages typically support 1 byte chars: enough to represent ASCII. More recently (Java) 2 bytes: for Unicode.

### 3. INTEGER Typical Representation: 1 word

Issues:

- length (precision)
  - fixed by lang. *specification? implementation?*
  - lang. spec. supports *multiple* lengths (C [`int`, `long`, Fortran, and esp. Java)
- sign: unsigned integers in C, Modula-2 ("natural nrs")

*non-portable code*



### 4. FLOAT/"REAL"-- same issues as for integers

Representation: <sign s><mantissa m><exponent e>

*Single precision reals (C float):* s is 1 bit, m -- 23 bits; e -- 8 bits. NOTE: e=0 represents exponent -127, to avoid exponent sign.

*Double precision (C double):* m -- 52 bits, e -- 11 bits.

Recall: float pt. ops are more expensive in hardware.

## More 'discreet' types

### 5. FIXED POINT TYPE -- in Ada (Cobol,PL/I)

- has implied decimal point: `fixed-decimal(5,2)`  
enough for \$0.00 to \$999.99; good when limited range of values; avoid extra cost of float point (e.g., on a 32-bit machine, with 2 decimals can represent up to 1 billion in one word; would require *double precision float* to represent the same value precisely)

### 6. ENUMERATION: (small) ordered set of named elements

Pascal, and descendants: `type Color=`

```
{red,orange,yellow,green,b,i,v}
```

C: `enum Day{mon,tue,wed,thu,fri,sat,sun}`

Operations:

comparisons: `(red < green)`

'succesor/previous': `(orange == succ(red) )`

conversion: `ordinal(orange) = 2`

Issues: Can same element name appear in more than one enumeration type (*overloading*):

```
enum Light{red,yellow,green}
```

C, Pascal: **no**

Ada: **yes, but** context must be able to disambiguate

Implementation:

- usually successive integers;

C: this shows through because the above is identical to

```
typedef int Day; const mon=0;tue=1;...
```

so type system allows comparing,adding weekdays and colors. (coercion, rather than explicit conversion:- ( )

- Ada and C allow over-riding default:

```
enum specialRegisters{gp=28,fp=30,sp=29}
```

# Subranges

**SUBRANGE TYPE:** a type whose values come from a *contiguous* subset of the values of some **base** type. Usually used with discrete types.

**INTEGER SUBRANGES:** Pascal and descendants:

```
type Age = 0..120; //very useful for arrays
```

**SUBRANGES OF OTHER BUILT IN TYPES:**

```
type UpperCase = 'A'..'Z';
```

**ENUMERATION SUBRANGES:**

```
type weekend_Day = {sat, sun};
```

Issues:      *Type checking:*

```
weekend_day d; ... d=succ(d);
```

```
Var mine,yours: Age; ...
```

```
  read(mine); yours= mine + 4;
```

- *Requires run-time checks.* So PLs supporting subranges require *some* run-time checking. (Language design trick: define subranges as *runtime constraints*, not types, so you can claim to be strictly statically checked. Runtime constraints are *generally* useful:

- C, Eiffel, Clu - `assert(<condition>)`

can check arbitrary constraints, but can be turned off, for efficiency. (A.C.Hoare: "this is like practicing sailing in harbour with life jacket, but then leaving it behind when sailing out on the ocean")

- **SUBTYPING:** if a function F or variable expects a value of type `Day`, it should accept one of type `weekend_Day`. A safe form of coercion.

Representation:

Usually the same as for integers, so often wasted space (e.g., for Ages, only need 7 bits)

# Structures/Records

Heterogeneous collection of labelled values.

Algol-like:

```
type Person = record  
  name:String(10);  
  age:integer  
end;
```

In C: typedef struct pers\_tag  
 { char\* name; int age; }  
 CPerson;

"struct pers\_tag" but not "pers\_tag" is a type in C.

Operations:

- accessing field to store and retrieve value; dot not'n  
`Person p, q; p.age = p.age + 1;`
- assigning entire record: `q=p;`
- comparing for equality: `if(q==p)`

Implementation:

- Usually successive memory locations, accessed by *displacement* from the start;

`CPerson: |< for char*> |<for int > | ~ 1word + 1 word`

So `p.age = displacement 1`; `p.name=displacement 0`;

(This is why *order* of fields matters most in C)

- **BUT:** have to worry about alignment of addressing:  
memory addresses for int's are word aligned (divisible by 4); for char's it is byte aligned (divisible by 1);

So we need padding:

`Person: |<10 chars for name> |xx|<int for age> |`

`name` starts at location `s`;

`xx` begins at `s+10*1byte (=s+2.5 words)`

`age` begins at `s+3 words`

# Unions

**Heterogeneous *alternative, mutually exclusive* values.**

(should be able to share storage)

Example variable declarations:

Algol68: `union(int,real) irx;`

C: `union Data {int i; double d;} idy;`

Operations:

- assignment L/R-value:

```
idy.i = 3; idyy=idy; idyy.i+=2;
```

Type checking problem:

```
if (n>0) {idy.i = 3;};  
else {idy.d = 3.45;};  
idy.d=3.45; //is this ok?
```

Requires i) a separate field for every union variable, which indicates type of value stored in it *now*; ii) *run-time* type check whenever union vars involved; no possibility of static guarantee.

[C does not do this!!! One reason why C not type-safe.]

Because of this most languages avoid it. Algol68 does allow static type checking using the following idea: the *only* way to access a value of union type is through a *case* statement:

```
case irx in  
  (int i): i+=1;  
  (real r): r=r/2;  
esac; //must cover all cases in union
```

Implementation:

This is the whole point: find storage need of largest value in union; the others can be overlaid in initial part; as mentioned, for type safety, need hidden "tag field".

## Variant records

**Observation: unions are used inside records**

*"persons have name, age, and either army service period (for males) or number of children given birth to (for females)"*

Pascal:

```
type Gender = {male,female};  
var me : record  
    name : String;  
    age : integer;  
    case sex : Gender of  
        male: (service : real); //years  
        female: (nr_of_children : int)  
    end //of person with variant
```

**sex** is also a field of the record, called a *discriminant* since its value determines which other fields are present; operations:

```
me.sex=male; me.service=2.5    or  
me.sex=female; me.nr_of_children+=1
```

Type checking: should not allow

```
me.sex=male; ... me.nr_of_children+=1; nor  
me.sex=male; me.service=2.5; me.sex=female
```

- Unfortunately, Pascal does not check correlation (so not type safe); Pascal, C do not even require the discriminant field -- it is up to you to use everything properly.
- Solution: - i) accessing field only allowed if discriminant appropriately set (simple dynamic check); ii) changing discriminant must go hand in hand with assigning value to coresp. variant

Implem'tn: variable part at the end, overlaid as record unions