

Types

- **What are types? (intro)**
- **The benefit of types**
 - for users
 - for implementers
- **Type systems: rules for**
 - primitive and constructed
 - assigning types to expressions
 - type compatibility
 - type conversion
- **Type checking**
 - strict vs. non-strict
 - run-time/dynamic vs. compile time/static

What is a type?

“A set of values and operations involving them”

- **Integers:**

`+ - * % > ==`

- **Arrays:**

`lookUp(<array>,<index>),
assign(<arr>,<indx>,<val>),
?initialize(A), ?setBounds`

- **User-defined types:**

`Java interfaces`

Utility of types for PL *users* (=programmer)

“A well-typed program does not go ‘wrong’ ” e.g., a function is never applied to the wrong type of argument. **Adds an extra level of safety beyond syntax checks:**

- **Analogy: UNITS in physics** (recall $distance\ fallen = 0.5 * g * t^2$)
 - if you want to find out the distance fallen by some object in 3 minutes, and you write $0.5 * 9.8 * 3^2$, you can tell you are wrong if you add units: $0.5 * 9.8m/sec^2 * 3min^2$ yields a value with unit $(m\ min^2)/sec^2$ which does not simplify to meters.
 - if you add a conversion factor of $60\ sec/min$, the units work out because $(m/sec^2) (min * sec/min)^2$ does cancel out meters!

NOTE:

1. there are rules for assigning units to expressions (the simplification rules, the unit conversion rules)
2. units cannot save you from *all* mistakes: e.g., if you wanted to know the distance fallen on the moon (need different value for g !!!)

Utility for programmer (more)

Programmer can ‘customize’ the language to the domain

PrimaryColors are {red,blue,green} -- enumeration type

a Student has Name and Id#, -- record type

a Roster is a List of Students -- list type

Utility for PL implementer

Associate storage requirements to types, and so assist with storage allocation.

Headaches of types

- **for programmer**

Must start somehow the process of typing

– usually need declarations for user-defined constants, variables, functions

w [e.g. procedural languages: C,C++,Pascal, Ada,...]

- **for PL implementer**

- Implementing type checking
- For implicitly typed languages, carrying around a type with (every/some) values at run-time

- **for PL designer**

- Balance all the above.

Mixed typing: implicit by default but requires occasional type specification by programmer to disambiguate

w [e.g. modern functional languages: SML, Haskell]

3 (complimentary) Views of Types

- **Set point of view:** e.g.,
 - *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*
- **Constructive point of view**
 - **Primitive types** e.g., *int, char, bool, enum{red,green,yellow}*
 - **Composite/constructed types:**
 - array e.g., *arrayOf(char)*
 - reference e.g., *pointerTo(int)*
 - record/structure e.g., *record(age:int, name:string)*
 - union e.g. *union(int, pointerTo(char))*
 - discriminated union/record variants
e.g., *union(male: record(armyService:bool,...),
female: record(numbrOfBirths: int))*

What is a Type System--more details on next slides

1. Rules for constructing types

- see previous page

2. Rules for determining/infering the type of expressions

- based on “fundamental rule”:

When function f has type *domain* S and *range* T ($S \rightarrow T$)
and expression e has type S , $f(e)$ has type T

3. Rules for type compatibility:

- In what contexts can values of a type be used (e.g., assignment, arguments of functions,...)

4. Rules for type equivalence, conversion

- needed for determining (ensuring) that an expression can be used in some context

Some terms about type systems

- **TYPE ERROR:** occurs when a program uses a value outside its proper “context”
- **TYPE CHECKING:** ensuring no type errors
 - **Type safe program:** runs without type error
 - **Type safe language:** allows execution of only type safe programs
 - **Strongly typed language:** PL does not allow unsafe programs by definition (i.e., enforces type safety by type checking --)
 - w *[assembler lang. is not type safe]*
 - w *[C has some unsafe parts]*
 - When determined:
 - w **Statically, at compile time**
 - w **Dynamically, at runtime**
 - w **Mixed**
 - How determined
 - w **Explicitly typed:** programmer makes declarations
 - w **Implicitly typed:** compiler/interpreter infers types

Determining Type of Expressions

Fundamental rule:

If function f has type $S \rightarrow T$ and expression e has type S , then $f(e)$ has type T

- 3.5 has type *float*; `round` has type *float* \rightarrow *int*; so `round(3.5)` has type *int*;
- 3 and 2 have type *int*, and `div` has type *int* * *int* \rightarrow *int* ; therefore the type of `(3 div 2)` is *int*

Can also be used backward: the type of arguments is inferred from the type of the function

- since `round` has type *float* \rightarrow *int*, the expression `round(x)` has type *int*, and x must have type *float*, if there are to be no errors.

Type error - using wrongly typed operands in an operation

- `3.5 div 2`
- `"abc"+ round(x)`

Type Checking

- ***Goal:*** to find out as early as possible, if each procedure and operator is supplied with the correct type of arguments
- ***Compile-time (static)***
 - **At compile-time, associates a type with every expression, using declaration information or types inferred from variable uses.**
Modern PLs often designed to do type checking (as much as possible) during compilation
- ***Runtime (dynamic)***
 - **During execution, check type of argument before doing operations**
Uses type tags to record type of value stored in variables at run-time
- ***Mixed***
 - Most type checking at compile time, but occasional run-time constraint needs to be verified (e.g., index of arrays is within its bounds)

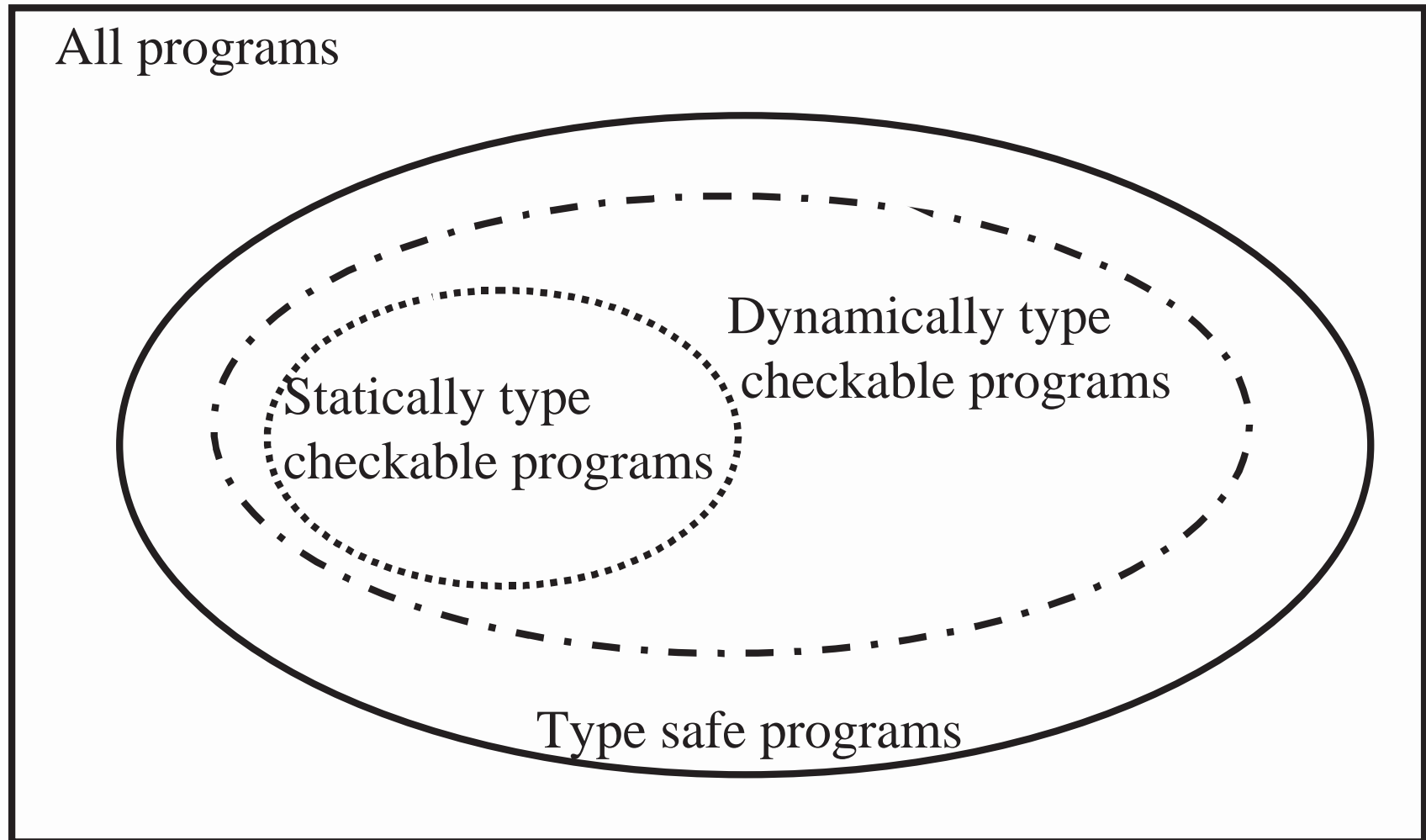
Type Safety & Strong Typing

- **A *type safe* program executes on all inputs without type errors**
 - Type safe depends on what are types, and does not mean “without errors”
- ***Strongly typed PL*: requires all programs to be type checkable (requirement on compiler)**
 - ***Statically strongly typed PL*** -
compiler allows only programs that can be type checked fully at compile-time. Some type safe programs cannot be statically typed:

```
if n=0 then y:="ab";  
else if n=0 then x := 'a'-5;
```

the unsafe part --assignment to x -- will never be executed, but no type checker can determine this.
 - ***Dynamically strongly typed PL*** -
Operations include code to check runtime types of operands, if type cannot be determined at compile-time

Hierarchy of Programs



Type Checking

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

	<u>static</u>	<u>dynamic</u>	<u>some dynamic</u>
<u>implicit</u>	SML	Scheme	
<u>explicit</u>	Ada		C, Pascal

Type Conversion

- **Explicit** -- using special functions
 - In Pascal, can explicitly convert types which may lose precision (*narrowing*)
 - `round(s)` real \rightarrow int by rounding
 - `trunc(s)` real \rightarrow int by truncating
 - In C, casting sometimes is explicit conversion
 - `sqrt((double) n)` where n is declared to be an `int`
 - In Ada, Java, abstract type may provide its own converter(s)
- **Implicit** (“coercion”)
 - e.g., in C, mixed mode numerical operations
 - `double d, e; ...e=d+2; //2` coerced to 2.0
 - PL/I problem: $(d \leq e \leq f)$ does not signal type error because $(d \leq e)$, of type *Bool*, is coerced into the type of `f` (`int`?) and then compared
 - Usually can use *widening* or conversion without loss of precision
 - integer \rightarrow double, float \rightarrow double
 - But real \rightarrow int may lose precision and therefore cannot (should not) be *implicitly* coerced!
 - Cannot coerce user-defined types or structures

Type Equivalence

If context (e.g, procedure, L-value) requires a certain type T, when can an expression of type T' be used there (e.g., as argument, R-value).

Why isn't this trivial?

```
typedef int dollars;
typedef int francs;
dollars x,w;    francs y;    int k;
    x=2;  w=x;    //ok
    y=w;  y+=k;   //want type error!!
ListOf(int) a;  ListOf(int) b;  ListOf(francs) c;
    a=b;        //ok
    c=b;        //type error
Struct{f:int; g:char;} d;
    d = {f=2,g='a'};           // ok
    d = {fh=2, gh='a'};       //??
    d = {2,'a'};              // now?
    d = {g='a', f=2;};        // still ok?
    d = {'a', 2};             //probably not. Why?
```

1. ‘Structural’ Equivalence

- “same shape” type in user-defined types:
 - type S = array [0..99] of char
 - type T = array [0..99] of char
 - typedef struct {
 int j, int k, int *ptr} cell;
 - typedef struct {
 int n, int m, int *p} element;
- **Given:** T as above,
 x,y: array [1..20] of int; *x,y,w,z,v are all of the same type*
 w,z : T; v: T; *by structural equivalence*

Some rules for structural type equivalence (Sethi)

- **SE1:** a type name is structurally equivalent to itself
- **SE2:** two types are structurally equivalent if they are formed by applying the same type constructor to two structurally equivalent types
- **SE3:** after type declaration `type id=T` the type name `id` is structurally equivalent to `T` <messes up dollars and francs>
- (**SE4:** if T1 equiv. T2, and T2 equiv. T3, then T1 equiv. T3)

2. Standard 'Name' Equivalence

Types are equivalent when they have the same name or are formed by the same type constructor. (Tries to preserve difference between dollar and francs where `typedef float dollar;` and `typedef int francs;`)

Given declarations:

```
T: type array [1..20] of int;  
x,y: array [1..20] of int;  
w,z : T;  
v: T;
```

x,y are of the same type, as are w,z,v; but x and w are different

C/C++ type system: structural equiv. for all types except structs

```
type A = record  
    x,y : real;  
end;  
type B = record  
    z,w : real  
end;
```

In C, A and B are not equivalent types because the fieldname is part of the type.

Other variants of type equivalence

- **Declaration equivalence (SE1 only)**

x, y : array[1..20] of int;

x, y are type equiv.

w: array[1..20] of int;

x, w are NOT type equiv

(has happened in languages where compiler actions defined type!)

- **Weak name equivalence (SE1, SE2 when both ids)**

typedef T int;

typedef S T;

both are equivalent

- **Expression equivalence (SE1, SE2 only)**

when you introduce a name, you break the chain of equivalence