Typing

Free

Strong

Wak

\text{e}
Well-typed programs cannot “go wrong.”

Overview of Types

- Data representation
- Type checking, strong typing, type inference
- Type insecurities in PL/I, Pascal
  - Ada subtypes
- Type equivalence
  - Ada derived types
- Polymorphism [Webber, Chapter 8]
  - Cardelli-Weger taxonomy

Universal parameteric polymorphism is important in ML, Haskell (later).
Data

- Raw (uninterpreted): words, bytes, bits
- Numbers: integers (excess, twos-complement), floating-point (IEEE 754, IBM) characters (ASCII, Latin-1)
- Multimedia: JPEG, GIF, WAV, MIDI, MPEG
- Programs
IEEE 754

**Single precision**

<table>
<thead>
<tr>
<th>s</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
</table>

Exponent bias $b = 127$

**Double precision**

<table>
<thead>
<tr>
<th>s</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
</table>

Exponent bias $b = 1023$

<table>
<thead>
<tr>
<th>e</th>
<th>f</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>±0</td>
</tr>
<tr>
<td>$2^b + 1$</td>
<td>0</td>
<td>±∞</td>
</tr>
<tr>
<td>$1 \leq e \leq 2^b$</td>
<td>&lt;any&gt;</td>
<td>$\pm 1 \cdot f \times 2^{e-b}$</td>
</tr>
<tr>
<td>0</td>
<td>$\neq 0$</td>
<td>$\pm 0 \cdot f \times 2^{1-b}$</td>
</tr>
<tr>
<td>$2^b + 1$</td>
<td>$\neq 0$</td>
<td>NaN</td>
</tr>
</tbody>
</table>

**Figure 5.5** The IEEE 754 floating-point standard. For normalized numbers, the exponent is $e - 127$ or $e - 1023$, depending on precision. The significand is $(1 + f) \times 2^{-23}$ or $(1 + f) \times 2^{-52}$, again depending on precision. Field $f$ is called the fractional part, or fraction. Bit patterns in which $e$ is all ones (255 for single-precision, 2047 for double-precision) are reserved for infinities and NaNs. Bit patterns in which $e$ is zero but $f$ is not are used for denormal (gradual underflow) numbers.
## IEEE 754

Standard for 32-bit floating-point number:

<table>
<thead>
<tr>
<th></th>
<th>$e$</th>
<th>$f$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Zero</strong></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>Infinity</strong></td>
<td>$2b+1$</td>
<td>0</td>
</tr>
<tr>
<td><strong>Normalized</strong></td>
<td>$1 \leq e \leq 2b$</td>
<td>$\neq 0$</td>
</tr>
<tr>
<td><strong>Denormalized</strong></td>
<td>0</td>
<td>$\neq 0$</td>
</tr>
<tr>
<td><strong>NaN</strong></td>
<td>$2b+1$</td>
<td>$\neq 0$</td>
</tr>
</tbody>
</table>

where $b$ is the exponent bias. For single precision, $e$ is 8 bits ($b = 127$), and $f$ is 23 bits. For double precision, $e$ is 11 bits ($b = 123$), and $f$ is 52 bits,
IEEE 754

Decimal number standard for currency
Due to the importance of decimal arithmetic in commercial applications and the potential speedup achievable], microprocessors supporting decimal floating-point (DFP) arithmetic are now available. IEEE 754 specifies three standard floating-point decimal data types of different precision:
Decimal32 floating-point format Decimal64 floating-point format
Decimal128 floating-point format
Examples are the decimal.Decimal type of Python, BigDecimal class in Java, and built-in type ’decimal’ in C#.
At what point do these two summations differ from the mathematical ideal?

```java
long i=0;
for (long j=0; ; j++) i = i+1;

float f=0.0f;
for (long j=0; ; j++) f = f+1.0;

for (long j=Long.MAX_VALUE-3, j<=LONG.MAX_VALUE; j++) {
    System.out.println (j);
}
```

The finite representation of numbers as bit patterns in a computer word require the program to understand the properties of these computer-represented numbers.
Interpretation

The same internal representation (bits) can mean different things. Take, for example, the 32 bits

\[ 0x9207BFF0 = 1001\ 0010\ 0000\ 0111\ 1011\ 1111\ 1111\ 0000 \]

These bits could mean different values depending on the interpretation.

-1844985872 # twos complement
2449981424 # unsigned integer
-4.283507E-28 # IEEE 754 floating point
add %fp, -16, %o1 # SPARC assembly code
transparent blue # alpha RGB (Java color)
A *data type* is a description of a collection of data values. The language implementer needs to know how to represent the values to implement the operations of the language. Often values of different data types have the same representation; often data (bit patterns) do not correspond to any value of the type. Presumably, no operation would purposely produce the wrong result on legitimate input. But bad things happen if it is possible to misinterpret the bits. If you assume the bits are a legal value, you may get unpredictable behavior.

How is a program to know if the programmer has called for an operation to be applied to values of the wrong type? (Can’t tell, in general; consider division by zero, it’s undecidable.)
Type insecurity arises when the data is misinterpreted. Type checking is verifying that the data types of actual parameters are appropriate for the operation, i.e., bits won’t be misinterpreted. Static type checking is type checking done by the compiler as opposed to being done at run time. Strongly typed language is one in which no misinterpretation of bits occurs at run time and all type checking is done at compile time. This requires careful attention the definition of data type. Significance: types at compile approximate the runtime values and type checking is a coarse verification of the program. Discovering the types of identifiers without declarations in the program is type reconstruction.
A language is said to be *type complete* if all the objects in the language have equal status. In some languages objects of certain types are restricted. For example, in Ada it is not possible to pass objects of function types as parameters, but one can pass records and arrays. In Pascal, for example, it is not possible to declare variables of function types. Functions usually have inferior status in ALGOL-like languages.
Type System

A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute.

Pierce, *Types and Programming Languages*, page 1.
The syntactically well-formed expressions are not uniformly “good;” some are badly behaved or ill-typed. All well-typed expressions are “good.” But not all good and useful expressions are well-typed. Some well-typed expressions, for example, $5/x$, are not perfectly good. The definition of “good” must be carefully considered to include as many good things as possible and exclude as many bad things as practical.
In some contexts tractable could be mean polynomial time (and not exponential).
Curiously, some compiler algorithms are exponential (ML type checking), but are deemed tractable in practice.
Type Systems

The system of types in a strongly-typed programming language is a compromise between verifying all the properties of data that the programmer would wish and what is decidable. A major challenge in programming language design is creating a type system with the richest collection of expressible properties yet such that the compiler can still guarantee that no data type description will be misinterpreted at run time.
well-typed expressions

bad expressions

all expressions

good expressions
The program computes with words; its upto the programmer to maintain the interpretation.
Not Strongly-Typed Languages

- Perl – name used three ways: scalars (string or numbers) begin with $, arrays begin with @, tables begin with %.
- ICON
- APL – scalar, vector, 2D array, …
- LISP – atoms, lists, lists of lists

Cost: errors not detected, type-checking at run time (e.g., addition of integers versus real, car of atom, etc.)
Types Are Distinct

- Integer
- Float
- Boolean
Mathematicians

Mathematicians usually conserve notation (symbols) and expect the human to disambiguate. Consider: natural numbers, integers, real numbers, complex numbers.

Real numbers represented by computers do not have nice mathematical properties and hence are really a completely separate type.

Another example, is \( c \) a symbol or regular expression?
Kinds of Types

- primitive, basic, unstructured — integer, real, boolean
- composite, structured — arrays, records, function types

The boundary is not always clear: is the string data type primitive or composite, for example. What is pointer?
Record types allow related data of heterogeneous types to be stored and manipulated together. Originally introduced by COBOL, records also appeared in Algo 68, which called them structures, and introduced the keyword struct. Many modern languages, including C and its descendants, employ the Algo terminology. Fortran 90 simply calls its records “types”: they are the only form of programmer-defined type other than arrays, which have their own special syntax.
Composite Types

<table>
<thead>
<tr>
<th>homogeneous</th>
<th>heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>static</td>
<td>record</td>
</tr>
<tr>
<td>dynamic</td>
<td>array</td>
</tr>
</tbody>
</table>

Associative arrays (or tables, hashes) as in Perl or Icon. (Sebesta, 5.6 Associative Arrays.)

```perl
$days{"Jan"} = 31;  # grows dynamically
foreach $item (%days) {
    print "The number is: $item.\n";
}
```

Often implemented using hash tables
Arrays

Slices and array operations; Scott 7.4.2 Dimensions, Bounds, and Allocation.

http://www.cs.rochester.edu/u/scott/pragmatics/figures/chap07.html#F07.10
# Procedure Types

<table>
<thead>
<tr>
<th></th>
<th>ML</th>
<th>Haskell</th>
<th>ANSI C</th>
<th>ALGOL 68</th>
<th>Modula-3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Procedure Type</td>
<td>int*real-&gt;bool</td>
<td>(Int, Float)-&gt;Bool</td>
<td>int f (int x, float y)</td>
<td>proc (int, real) bool</td>
<td>PROCEDURE (x:INTEGER, y:REAL): BOOLEAN</td>
</tr>
<tr>
<td></td>
<td>ML</td>
<td>Haskell</td>
<td>ANSI C</td>
<td>ALGOL 68</td>
<td>Modula-3</td>
</tr>
<tr>
<td></td>
<td>int-&gt;unit</td>
<td>Int-&gt;()</td>
<td>void f (int x)</td>
<td>proc (int) void</td>
<td>PROCEDURE (x:INTEGER)</td>
</tr>
</tbody>
</table>

A procedure type is needed in programs that pass procedures as arguments. Ada example:

```
funarg/trapezoid.adb
```
Data Abstraction

Good design enables easy handling and high volume
Constructors of Structured Types

The value of composite data types (records, arrays, functions) is to elevate computation to a more abstract world of more meaningful computational values.
Constructors of Structured Types

It is possible in Ada, and some other languages, to construct arrays and records all at once, and not just piece by piece. The expressions that use constructors are called *aggregates* in Ada.

```ada
type Date is
  record
    Day: Integer range 1 .. 31;
    Month: Month_Type;
    Year: Integer range 0 .. 4000;
  end record;

A: Date := Date'(4, July, 1776); -- positional association
B: Date := Date'(Day=>4, Month=>July, Year=>1776);
C: Date := Date'(Month=>July, Day=>4, Year=>1776);
D: Date := Date'(2+2, Month_Type'Succ(June), 3552/2);
```

Treating data as a unit is the hallmark of data abstraction.
Features of Ada Type System

We can illustrate a large number of type issues using Ada.

1. Strongly typed, name equivalence, no implicit coercions, overloading
2. Subtypes — compile-time verification versus run-time checking
3. Derived types — overcoming limitations of name equivalence
4. Tagged records — subtyping
5. Generics — parametric universal polymorphism
6. Packages — type abstraction
Information about Ada


Examples of Type Insecurities

1. PL/I — pointer to “anything”
2. Pascal
   2.1 Subranges
   2.2 Variant records
   2.3 Subprocedures as arguments
3. Russell’s paradox
Obviously, PL/I’s “pointer to anything” is insecure.

POINT:    PROCEDURE OPTIONS (MAIN);
    DECLARE
        1 NODE BASED (P),
        2 INFO FIXED,
        2 LINK POINTER,
        1 STUDENT BASED (Q),
        2 NAME CHARACTER (30),
        2 GPA FLOAT,
        X POINTER;
    ALLOCATE NODE; /* P points to new record object */
    ALLOCATE STUDENT; /* Q points to new record object */
    X = P; /* X points to node */
    X->GPA = 3.75; /* A node does not have a GPA! */
END POINT;
Type Insecurities of Pascal

1. Subranges
   Ada uses *runtime* solution

2. Variant records
   Euclid has a solution, Ada uses a compromise

3. Subprocedures as arguments
   Fixed in ISO 7185 Pascal, 1983.
Type Insecurities of Pascal

(* 11111111111111111111111111111111 (*)

var
  wide: 1..100; narrow: 10..20; farout: 150..300;
begin
  narrow := farout; wide := narrow; narrow := wide
end
Subranges in Ada

Subtype. A **subtype** in Ada is *not* a new type. Rather it is a constraint on the values enforced at run time. If a value does not satisfy the constraint at runtime, the exception **CONSTRAINT_ERROR** is raised.

```ada
declare
    subtype Wide_Type is Integer range 1..100;
    subtype Narrow_Type is Integer range 10..20;
    subtype Farout_Type is Integer range 150..300;
    Wide: Wide_Type;
    Narrow: Narrow_Type;
    Farout: Farout_Type;
begin
    Narrow := Farout;  Wide := Narrow;  Narrow := Wide;
end;
```
Subtypes in Ada

- Character
  - range 'A'..'Z'

- String
  - strings of length 10
  - strings of length 5

- Boolean

- Float
  - digits 3
  - range 0.0..1E6
  - digits 4
  - range 0.0..1E5

- Integer
  - range -50..25
  - range 0..100
  - range 1..500

- All other types
Hoare in his 1981 Turing award lecture decried the lack of runtime range checking. The problem persists today.

In any respectable branch of engineering, failure to observe such elementary precautions would have long been against the law.
Figure 1. The number of software vulnerabilities cataloged by the NIST National Vulnerability Database skyrocketed in 2017, and the fraction of vulnerabilities involving buffers (either categorized as “buffer error” or containing the keyword “buffer”) kept pace.
Design & Implementation

Stack smashing

The lack of bounds checking on array subscripts and pointer arithmetic is a major source of bugs and security problems in C. Many of the most infamous Internet viruses have propagated by means of stack smashing, a particularly nasty form of buffer overflow attack. Consider a (very naive) routine designed to read a number from an input stream:

```c
int get acct num(FILE *f) {
    char buf[100];
    char *p = buf;
    do {
        /* read from stream s: */
        *p = get c(s);
    } while (*p++ != ' \n');
    *p = ' \0';
    /* convert ascii to int: */
    return atoi(buf);
}
```

If the stream provides more than 100 characters without a newline (\n), those characters will overwrite memory beyond the confines of buf, as shown by the large white arrow in the figure. A careful attacker may be able to invent a string whose bits include both a sequence of valid machine instructions and a replacement value for the subroutine’s return address. When the routine attempts to return, it will jump into the attacker’s instructions instead.

Stack smashing can be prevented by manually checking array bounds in C, or by configuring the hardware to prevent the execution of instructions in the stack (see the sidebar on page ©179). It would never have been a problem in the first place, however, if C had been designed for automatic bounds checks.
What are variant records?
Out of fashion because of subclasses in OO languages.

- Euclid – type safe
- Ada – runtime checking

variant/vr.adb
Type Insecurities of Pascal

Two interpretation of the same bits.

(* 2222222222222222222222222222222222222222222222222 *)

```pascal
type
  option = (a, b);
  vrt = record case tag: option of
    a: (f1: real);
    b: (f2: integer)
  end

var x: vrt
begin
  x.f1 := 12.65;
  x.tag := b;
  if x.f2 = 32 then (* ... *)
end
```
Euclid

var x : vrt(a);
var y : vrt(b);
var z : vrt(any);

{ One can assign 'x' to 'z', but not vice versa. }
z := x;

{ Discriminating analysis of z: }
case discriminating w = z on tag of
  a => { use w as if it were declared vrt(a) }
  b => { use w as if it were declared vrt(b) }
end case
Type Insecurities of Pascal

Procedure arguments declared without argument types.

\[
P (x: \text{integer}; \text{procedure } F; y: \text{real});
P (x: \text{integer}; \text{function } F: \text{real}; y: \text{real});
\]

```pascal
program Main (input, output);
procedure Print (x: integer; procedure P);
begin
  if x=1 then P (2.1) else P (3.2, 9.3);
end { Print };
procedure Print1 (x: real); ...
procedure Print2 (x,y: real); ...
begin { Main }
  Print (1, Print1);  Print (2, Print2);
end.
```
Type Insecurities of Pascal

Procedure arguments declared with argument types.

\[
P (x: \text{integer}; \text{procedure } F(a: \text{real}; b: \text{char}); y: \text{real});
\]

\[
P (x: \text{integer}; \text{function } F(c: \text{real}): \text{real}; y: \text{real});
\]

```
program Main (input, output);
    procedure Print (x: integer; procedure P);
    begin
        if x=1 then P (2.1) else P (3.2, 9.3);
    end { Print };
    procedure Print1 (x: real); ... 
    procedure Print2 (x,y: real); ... 
    begin { Main }
        Print (1, Print1); Print (2, Print2);
    end.
```
Type Equivalence

Bessie, Buttercup, Rossie, MooMoo, MooLawn, MosCow, ...
Type Equivalence

Type equivalence or compatibility. Type equivalence concerns when types are considered equal in a programming language for purposes of assigning a value of one type to a location that can hold another type, or of passing a value of one type to a subprocedure expecting a parameter of another type.

- name equivalence – types with the same name are equal.
- structural equivalence – types with the same underlying structure are equal

Sebesta 11th, Section 6.14, Type Equivalence.
Watt, Section 2.5.2, Type Equivalence.
Scott 4th, Section 7.2.1, Type Equivalence.
Structural equivalence requires a clever algorithm

type T1 is record
   X: Integer;
   N: access T1;
end record;
Structural equivalence requires a clever algorithm

type T1 is record
  X: Integer;
  N: access T1;
end record;

type T2 is record
  X: Integer;
  N: access T2;
end record;

type T3 is record
  X: Integer;
  N: access record
    X: Integer;
    N: access T3;
end record;

type T4 is record
  X: Integer;
  N: access T2;
end record;

type T5 is record
  N: access T5;
  X: Integer;
end record;

type T6 is record
  Y: Integer;
  N: access T6;
end record;
Structural equivalence requires a clever algorithm

type T1 is record
    X: Integer;
    N: access T1;
end record;

type T2 is record
    X: Integer;
    N: access T2;
end record;

type T3 is record
    X: Integer;
    N: access record
        X: Integer;
        N: access T3;
    end record;
end record;

type T4 is record
    X: Integer;
    N: access T2;
end record;

type T5 is record
    N: access T5;
    X: Integer;
end record;

type T6 is record
    Y: Integer;
    N: access T6;
end record;
Structural equivalence requires a clever algorithm

type T1 is record
  X: Integer;
  N: access T1;
end record;

type T2 is record
  X: Integer;
  N: access T2;
end record;

type T3 is record
  X: Integer;
  N: access record
    X: Integer;
    N: access T3;
  end record;
end record;

type T4 is record
  X: Integer;
  N: access T2;
end record;

type T5 is record
  N: access T5;
  X: Integer;
end record;

type T6 is record
  Y: Integer;
  N: access T6;
end record;
Structural equivalence requires a clever algorithm

type T1 is record
    X: Integer;
    N: access T1;
end record;

type T2 is record
    X: Integer;
    N: access T2;
end record;

type T3 is record
    X: Integer;
    N: access record
        X: Integer;
        N: access T3;
    end record;
end record;

type T4 is record
    X: Integer;
    N: access T2;
end record;

type T5 is record
    N: access T5;
    X: Integer;
end record;

type T6 is record
    Y: Integer;
    N: access T6;
end record;
Structural equivalence requires a clever algorithm

type T1 is record
  X: Integer;
  N: access T1;
end record;

type T2 is record
  X: Integer;
  N: access T2;
end record;

type T3 is record
  X: Integer;
  N: access record
    X: Integer;
    N: access T3;
  end record;
end record;

type T4 is record
  X: Integer;
  N: access T1;
end record;

type T5 is record
  N: access T5;
  X: Integer;
end record;

type T6 is record
  Y: Integer;
  N: access T6;
end record;
Type Equivalence

Are implementation difficulties an overriding concern?
Type Equivalence

Are implementation difficulties an overriding concern?

No, not really. But …
Type Equivalence

... structural equivalence spoils abstraction.

```
declare
type Point is
    record First, Second: Float end record;
type Complex is
    record First, Second: Float end record;
P: Point;
Z: Complex;
begin
    P := Z; -- P,Z hold elements of same structure
end;
```
Type Equivalence

By “name” equivalence we really don’t mean “same name,” but we mean referring to the same declaration. Clearly, the use of the same name in different scopes is unrelated.

```
declare
type Direction is (Up, Down);
D: Direction := Up;
begin
inner: declare
type Direction is (North, South, East, West);
E: Direction;
begin
-- Variables E and D do not have the same typ
-- even though the types have the same name.
E := D; -- illegal
end inner;
end;
```
Name equivalence: same name implies same type. The following is not legal Ada. There are no generative type declarations (they must be derived).

```
declare
    type Social_Security_Number is Integer;
    type Student_Identity_Number is Integer;
    SSN: Social_Security_Number;
    SIN: Student_Identity_Number;
    I: Integer;
begin
    SIN := 5; SSN := SIN; I := SSN;
end;
```
Name equivalence requires names (naturally), but sometimes the programmer would rather not have to invent another name.

```

type
T = record a: int; b: char end;
var
x, y: array[1..2] of record a: int; b: char end;
z: array[1..2] of T;
u, v: array[1..2] of record a: int; b: char end;

T = record a: int; b: char end;
Anon1 = array[1..2] of record a: int; b: char end;
Anon2 = array[1..2] of T;
var
x, y: Anon1;
z: Anon2;
u, v: Anon1;
```
Occasionally the programmer would like several names (aliases) for the same type. Perhaps because an “official” name is too long. Is it possible to get a type synonym in Ada?

```ada
type T is Integer; -- generative? Yes, new type.
```

Not in this way, type declarations in Ada are generative. Contrast this with Haskell

```ada
type T = Integer -- create alias
```

Synonyms can be achieved in Ada using subtypes with no constraints, as in the following example.

```ada
subtype T is Stacks_And_ Queues.Stacks.Internal_Repr
```
Ada Derived Types

Generative type declarations and name equivalence are good for keeping separate sets of values.

```
declare
    type Student_Id_Type is Integer;
    type Social_Security_Type is Integer;
    Id : Student_Id_Type;
    SSN : Social_Security_Type;
begin
    Id := SSN;  -- illegal, different types
    Id := Id + 1;  -- illegal, does not make sense
end;
```

But generative types lose all the operations and constants which apply without modification to the new type.
Ada’s derived types, which use the keyword `new` in the declaration, make a copy of the type including the operations.

```ada
declare
    type Social_Security_Number is new Integer;
    B: Social_Security_Number;
    I: Integer;
begin
    B := 5;       -- legal
    B := 2*B+5;   -- legal
    B := I;       -- illegal!
end;
```
Generative type declarations and name equivalence complicates similar types.

```plaintext
declare
  subtype Sub is Integer range 1..10;
  type Der is new Integer range 1..10;
N: Integer;
K: Der;
L: Sub;
begin
  null;
end;
```
Generative type declarations and name equivalence complicates similar types.

```这个世界
declare
    N: Integer;
    subtype Sub is Integer range 1..10;
    J : Sub;
    type D is new Integer range 1..10;
    K : D;
begin
    N := J;  -- legal
    K := K+1;-- legal
    J := J+1;-- legal
    N := K;  -- illegal
end;
```
Name Equivalence

Name equivalence does not provide abstraction (hence the need for the abstraction mechanisms to be discussed later under the topic of modules).

```plaintext
declare
type Point is record First, Second: Float end record;
type Complex is record First, Second: Float end record;
P: Point;
Z: Cx;
begin
  P := Z;  — illegal under name equivalence; not the same type, but the same effect can be obtained:
  P.First := Z.First;
  P.Second := Z.Second;
end;
```
Modula-3

Going against the general trend, the programming language Modula-3 uses structural equivalence.

1. Share complex data structures in distributed programs
2. Uses “brands” to create distinct types (with the same structure)
3. Subtyping is based on structure
Carambola or Starfruit?
Branding in Modula-3

```
MODULE Structure EXPORTS Main;

TYPE
  StarFruit = REF INTEGER;
  Carambola = REF INTEGER;
  Apple   = BRANDED REF INTEGER;
  Orange  = BRANDED REF INTEGER;
  Pepsi  = BRANDED "Pepsi" REF INTEGER;
  Coke   = BRANDED "Coke" REF INTEGER;
```

BEGIN
END Structure.

A brand makes the “name” part of the structure of the type. Using brands a programmer can achieve name equivalence for individual types.
Scott, 4th, Section 7.3 Parametric Polymorphism
## Cardelli-Wegner Taxonomy

<table>
<thead>
<tr>
<th>Polymorphism</th>
<th>Ad Hoc</th>
<th>Universal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implicit coercion</td>
<td>Overloading</td>
<td>Parametric</td>
</tr>
<tr>
<td>Inclusion</td>
<td>AKA Subtype</td>
<td></td>
</tr>
</tbody>
</table>
Polymorphism. A function is said to be polymorphic if it can take arguments of different types. Polymorphism can be divided into two kinds: *ad hoc polymorphism* and *universal polymorphism*.

*Ad hoc polymorphism* – a system in which a single function name may refer to any one operation from a finite number of possibilities, sometimes these possibilities are implemented with different code:

- implicit coercion – arguments converted to some type compatible with the function
- overloading – different functions of the same name
universal polymorphism – an infinite number of possibilities with common substructure treated uniformly
  ▶ parametric – implicit or explicit type parameter which determines type of argument to function
  ▶ inclusion – value can be viewed as belonging to multiple types
Milner (1978) gives credit to Strachey for “probably” being the first to call this flexibility “polymorphism” and coining the phrases “parametric polymorphism” and “ad-hoc polymorphism.”
Polymorphism vs Type Conversion

Polymorphism is *not* the same as type conversion. There are many functions that convert data of one type to data to another time. These conversions are not a mechanism of the language, but a capability of the built-ins, libraries, or the user. Here are some hypothetical examples:

\[
\begin{align*}
\text{real: } & \text{ int } \to \text{ float} \\
\text{floor: } & \text{ float } \to \text{ int} \\
\text{ceil: } & \text{ float } \to \text{ int} \\
\text{round: } & \text{ float } \to \text{ int} \\
\text{itoa: } & \text{ int } \to \text{ string} \\
\text{ord: } & \text{ char } \to \text{ int} \\
\text{char: } & \text{ int } \to \text{ char}
\end{align*}
\]

Some of these function may require many steps to implement, e.g., itoa, some may require no steps (the representation of the different data values may be the same), e.g., char.
Coercions

Many languages permit “mixed-mode” expressions:

822.34 + 4

Consider the expression $1/3 + 25$. In PL/I this expression has the value $5.33333333333$. If $n$ is an integer and the context requires a real number, then there is an obvious mapping that loses no information. Such coercions are called *widenings* or *promotions*. Going from a real number to an integer loses information, and there is more than one reasonable inverse mapping—rounding and truncating, in particular. Such a coercion is called a *narrowing*. 
Coercions

User-defined implicit type coercions in C++

coection/tiny.cpp
Coercions

Pre-defined implicit type coercions in C++

\[<< : \text{istream} \times \text{int} \rightarrow \text{istream}\]
\[<< : \text{istream} \times \text{float} \rightarrow \text{istream}\]

\[
\text{cin} \quad \text{<<} \quad \text{i} \quad \text{<<} \quad \text{f}
\]

while (\text{cin} \quad \text{<<} \quad \text{i})

\text{istream to boolean.}
Implicit Coercion

In C++, the compiler is allowed to make one implicit conversion to resolve the parameters to a function. What this means is that the compiler can use constructors callable with a single parameter to convert from one type to another in order to get the right type for a parameter.

One can prevent by using the C++ keyword `explicit`. 
Overloading

User defined overloading.

overload/example_1.adb
overload/example_2.adb
overload/largest.adb
Mixing Types Is Costly

In September 1999 NASA’s Mars Climate Orbiter crashed into Mars instead of going into orbit. Preliminary findings indicate that one team of programmers used English units while the other used metric units for a key spacecraft operation. Total mission cost of $327.6 million.

How can types be used to avoid this kind of problem?

In the Ada program overload/metric.adb, the arithmetic operators are overloaded to mix English and metric units correctly.

Dimensional analysis: “A Deeper Look at Metafunctions” by David Abrahams and Aleksey Gurtovoy at C++ source. Dimensionalized numbers an approach by Aaaron Denny using Haskell classes.
Overloading in C++

C++ does not allow overloaded functions that have the same number and types of arguments but differ only in their return value because C and C++ functions can be called as statements. When a function is called as a statement, ignoring the return value of the function, the compiler would not have any way of determining which function to call.

Mitchell, page 343.
Overloading and implicit coercion is confusing in C++.

```cpp
void f(int)
void f(int *)
f('a')
```

The call will result in `f(int)` because a `char` can be promoted to an `int`. 
Overloading

User-defined overloading in C++. C++ can overload operators. (Sebesta, Section 7.3 Overloaded Operators.) From Scott, Figure 3.19, page 146.

class X { /* ... */
    X& operator=(const X& from) { /* ... */ }
    Y& operator[](int) { /* ... */ }
    Z& operator+(int) { /* ... */ }
    Z& operator+() { /* ... */ }
};

Java decided not to permit overloading of operators.
C++ can overload operators. From Scott 4, Example 3.26, page 148–149.

```cpp
class complex {
    double real, imaginary;
    // ...

public:
    complex operator+(complex other) {
        return complex(real+other.real, imaginary+other.imaginary);
    }
    // ...

};
```

For complex x, y, z; we can write either of the two:

```cpp
z = x + y; // user-defined operator+
z = x.operator+ (y);
```
Universal polymorphism that takes types as parameters (arguments). Parameterized templates are macros for source code. An instantiation fills in the template with the actual time. Each instantiation can be compiled separately. For example, C++ templates, Ada generics. More clever approaches can use the same code for all types.

Do not confuse the creation of an instance of a class (new in Java), with the instantiation of an Ada generic packaged of function (also the keyword new in Ada).
generic
  type Type is private;
function Id (X : in Type) return Type;

function Id (X : in Type) return Type is begin
  return X;
end Id;

-- generic instantiations
function Int_Id is new Id (Integer);
function Float_Id is new Id (Float);
-- example uses
if (Int_Id (3) = 3) then
  X := Float_Id (3.4);
end if;
Ada Generics

-- generic function specification

generic
    type Item is private;
    with function "<" (X,Y: Item) return Boolean;
function Min (X,Y: Item) return Item;

-- generic instantiations
function String_Min is new Min (String, "<");
function Data_Min is new Min (Data, Date_Precedes);
-- generic function implementation
-- (cannot be combined with its specification)
function Min (X,Y: Item) return Item is
begin
  if X<Y then
    return X;
  else
    return Y;
  end if;
end Min;
Templates in C++

Mitchell, Section 9.4.3 C++ Function Templates, page 259ff

template<typename T>
void swap (T& x, T& y) {
    T tmp = x; x=y; y=tmp;
}

int i,j; swap(i,j);  // int for T
float a,b; swap(a,b);  // float for T
String s,t; swap(s,t);  // String for T

template<typename T>
void sort (int count, T* A[count]) {
    for (int i=0; i<count-1; i++)
        for (int j=i+1; j<count-1; j++)
}
Templates in C++

Sebesta, Section 9.8.2 Generic Functions in C++, page 383.

```cpp
template<class T> class vector { /* ... */ };  
template<class T> void sort(vector<T>) { /* ... */ }

vector<complex> cv(100);  
vector<int> ci(200);  
sort(cv); // invoke sort(vector<complex>)  
sort(ci); // invoke sort(vector<int>)  
```
Inclusion Polymorphism

The type $\tau$ can be used where the type $\sigma$ is called for. The type $\sigma$ is not just a single type, but it represents a set of types—all those than be used in its place. The type $\sigma$ includes all its subtypes. We write $\tau \leq \sigma$.

This has great intuitive appeal as in “every integer is a real number.” But this goes nowhere as a universal principle because in a computer “not every integer is a floating-point number,” and more over one cannot substitute the bit pattern for an integer in a machine floating opint operation.

But what common structure do some types have that allow this kind of substitution?
subtype polymorphism
aka inclusion polymorphism

\[ \sigma \leq \tau \]

a \( \sigma \) is a \( \tau \)
\( \sigma \) is a subtype of \( \tau \)
\( \sigma \leq \tau \)
Inclusion Polymorphism

{c:int}

{a:int, c:int}

{a:int, b:char, c:int}
class C {
    class Point { int x, y; }
    class Circle extends Point { int radius; }

    static move (Point p) { x += 1; }

    static public void main (String args[]) {
        Circle c = new Circle();
        move (c);
    }
}
Subtype Polymorphism

objects/shape_main.adb
Organize all the types into a partial order $\tau \prec \sigma$ such that the implementation will never care if when expecting a $\sigma$ an element of type $\tau$ is provided. The relation (a lattice) in which subtypes can be used without fear of type insecurity or knowledge of the implementation in place of the supertypes.

Easy: make all the types unrelated. Hard: find any polymorphism this way.

The solution comes in a surprising place: records.

For example,

\{k: \text{Int}, l: \text{String}\}

\{k: \text{Int}, l: \text{String}, m: \text{Real}\}
Subtype Polymorphism

{k: Int, l: String}

{k: Int, l: String, m: Real}  {k: Int, l: String, n: Char}
Inclusion Polymorphism

\[ \bot \]

\[ \{a:\text{int}, b:\text{char}\} \]

\[ \{b:\text{char}\} \]

\[ \{a:\text{int}, b:\text{char}, c:\text{int}\} \]

\[ \{b:\text{char}, c:\text{int}\} \]

\[ \{a:\text{int}\} \]
The type of a function $\tau = \tau_d \rightarrow \tau_r$ has a set of values of the domain $\tau_d$ and the range $\tau_r$. When is the type $\sigma$ a subtype of $\tau$? When is $\sigma \leq \tau$? Clearly $\sigma$ must be the type of a function, so, when is $\sigma_d \rightarrow \sigma_r \leq \tau_d \rightarrow \tau_r$?
Subtyping of Function Types

\[
\tau_d \quad \tau_r
\]
Subtyping of Function Types

\[ \tau_d \subseteq \tau_r \]

\[ \sigma_r \]

\[ \tau_r \]
Subtyping of Function Types

\[ \sigma_d \overset{\tau_d}{\hookrightarrow} \sigma_r \]

\[ \tau_r \overset{\tau_r}{\hookrightarrow} \tau_d \]
Subtyping of Function Types

This rule was first stated formally by John C. Reynolds,[Essence of Algol, 1981] and further popularized in a paper by Luca Cardelli [Semantics of Multiple Inheritance, 1984]. One can imagine the type of a function $\sigma_d \rightarrow \sigma_r$ having a set of values for the domain and for the range. What is the subtype of the function? (The subtype is also a function.) Arrow (function) types are contravariant in the domain and covariant in the range. If $\tau_r \leq \sigma_r$ and $\sigma_d \leq \tau_d$, then $\tau_d \rightarrow \tau_r \leq \sigma_d \rightarrow \sigma_r$. 
When one could not pass functions directly in Java, the subtype rules for functions is observable only with interfaces.
[The bat example??]
Now Java has lambdas.

```java
public interface Function<T,R> {
    R apply(T t);
}

class X implements Function<Bat,Food>{
    Vegetable apply (d: Mammal) {
        return new Vegetable();
    }
    void test (f: Function<Bat,Food) {
    }
    void main () {
        test (\m:Mammal -> new Vegetable());
    }
}
```

Functions are contravariant in the domain and covariant in the range.
Java and Array

Contrary to the laws of nature, arrays in Java are co-variant. (ArrayList is neither co- nor contra-variant as should be any mutable collection.)
Subtype Polymorphism and OO

Subtype polymorphism is related to OO, but how? Some important ideas in OO: inheritance, dynamic dispatch, nominal subtyping. More about OO later (if time).

objects/inherence.adb
objects/dynamic.adb
objects/overload.adb
class/Contra.java – covariance in the range
class/Vegetarian.java
Bounded Quantification

A hybrid form of polymorphism is found in recent versions of the programming language Java and C#.
We motivate the need by a series of examples in a hypothetical language:

```plaintext
fun f (x:{a:int}) = x; (* f:{a:int}->{a:int} *)

val ra = {a=0} (* record with "a" field *)
f (ra);
{a=0} : {a:int}

val rab = {a=0,b=true} (* record with "a", "b" field *)
f (rab); (* subtype polymorphism *)
{a=0,b=true} : {a:int}
```
fun fpoly ['X] (x:'X) = x;  (* f:'X->'X *)

fpoly [{a:int,b:bool}] rab;
{a=0,b=true} : {a:int,b:bool}

fun f2 (x:{a:int}) = {orig=x, s=x.a+1};  (* f2 : {a:int} -> {orig:{a:int}, s:int} *)

f2 ra;

{orig={a=0}, s=1} : {orig:{a:int}, s:int}

f2 rab

{orig={a=0,b=true}, s=1} : {orig:{a:int}, s:int}

f2poly ['X] (x:'X) = {orig=x,s=x.a+1};  (* type error *)

No way to constrain type of x to be a record containing a field "a".
The solution is a combination of parametric and subtype polymorphism.

\[
f2\text{poly } [\ 'X <: \ {a}\text{:int} ] \ (x:'X) = \ {\text{orig}=x, s=x.a+1};\]

\[
f2\text{poly} : 'X<:{a}\text{:int} -> \ {\text{orig}:'X, s\text{:int}}\]

\[
f2\text{poly} \ rab\]

\[
{\text{orig}={a=0, b=true}, s=1} : \ {\text{orig}:{a}\text{:int, b:boolean}, s\text{:int}}\]

Pierce, Chapter 26 Bounded Quantification.
Polymorphism in Java

- polymorphism/Example2.java
Polymorphism in Java

- polymorphism/Coercion.java
- polymorphism/Overloading.java
- polymorphism/Shapes.java — subtype polymorphism
- polymorphism/Parametric.java — universal, parametric polymorphism
- polymorphism/Wild.java — universal, parametric polymorphism
- polymorphism/Move.java — bounded polymorphism
- polymorphism/Example.java — bounded polymorphism
Java wild cards. Joshua Bloch calls those objects you only read from Producers, and those you only write to Consumers. He recommends: "For maximum flexibility, use wildcard types on input parameters that represent producers or consumers", and proposes the following mnemonic:

PECS stands for Producer-Extends, Consumer-Super.

Kotlin does not have Java’s wild cards. It uses See Generics in the Kotlin reference documentation.
Duck typing.
The name of the concept refers to the duck test, attributed to James Whitcomb Riley, which may be phrased as follows:

"When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck."

▶ polymorphism/DuckTyping.java – Via reflection
Summary of Types

- Data representation
- Type checking, strong typing, type inference
- Type insecurities in PL/I, Pascal
  Ada subtypes
- Type equivalence
  Ada derived types
- Polymorphism
  Cardelli-Weger taxonomy