Programming is not just managing the control flow (algorithm), but it is also the design of the abstract values upon which to compute (data structures).
Good design enables easy handling and high volume
Well-typed programs cannot “go wrong.”

Made key breakthroughs in disparate areas: proofs, languages, concurrency
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 parametric 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
Arecibo

Do all aliens speak English?

How would you try to communicate to an extraterrestrial civilization? Such a message was devised by Frank Drake and transmitted once in November 1974 toward globular cluster M13 from the Arecibo radio-telescope in Puerto Rico. The message consisted of 1679 bits.

(Hint: 1679 is the product of two primes 23 and 73.)
1. The top row represents the numbers from 1 to 10.
2. The cluster in the center (colored green) encodes the atomic numbers for hydrogen (1), carbon (6), nitrogen (7), oxygen (8), and phosphorus (15).
3. The next patterns (green) represent formulas for the four nucleotides of DNA.
4. The (while) vertical bar in the center specifies the number of base pairs in the human genome supposedly 4,294,441,822 (now estimated to be as high as 3.5 billion).
5. The double helix shape of the DNA molecule is represented by the (blue) curving lines that goes from the human figure.
The next set of symbols (white) represent the human population (on the right) on Earth—4,292,853,750, a figure of a human (red), and the height of a human, 14 units.

Our Solar System (yellow) is displayed next. The dot representing Earth is displaced toward the human being.

The Arecibo telescope dish (purple) is near the bottom of the picture. The last set of pixels (white) gives the diameter of the Arecibo Radio Telescope: 2,430 units of 12.6 centimeters or about 306 meters.
IEEE 754

Standard for 32-bit floating-point number:

<table>
<thead>
<tr>
<th></th>
<th>e</th>
<th>f</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Zero</td>
<td>0</td>
<td>0</td>
<td>±0</td>
</tr>
<tr>
<td>Infinity</td>
<td>2b + 1</td>
<td>0</td>
<td>±∞</td>
</tr>
<tr>
<td>Normalized</td>
<td>1 ≤ e ≤ 2b</td>
<td>0</td>
<td>±1.f × 2^{e-b}</td>
</tr>
<tr>
<td>Denormalized</td>
<td>0</td>
<td>≠ 0</td>
<td>±0.f × 2^{-b+1}</td>
</tr>
<tr>
<td>NaN</td>
<td>2b + 1</td>
<td>≠ 0</td>
<td>NaN</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

See IBM Packed Decimal Examples in programming languages are the decimal.Decimal type of Python, BigDecimal class in Java, and built-in type 'decimal' in C#.
Understand the data representation is (unfortunately) an indispensable part of programming, especially floating point numbers.
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 + 1.0f;
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
507,888E-65 # IEEE 754 decimal (BID)
add %fp, -16, %o1 # SPARC assembly code
transparent blue # alpha RGB (Java color)
Definition

A *data type* is a description of data values with abstract properties possibly including specific operations.

The language implementer needs to know how to represent the values to implement the operations of the language. The language implementer chooses the bit pattern for the data types.
Data Types

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 checking

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.
Type Completeness

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 (and pointers to 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.
Static Typing

Static types are the world’s most successful formal method. They allow programmers to specify properties of functions that are proved on every compilation. They provide a design language that programmers can use to express much of the architecture of their programs before they write a single line of algorithmic code. Moreover this design language is not divorced from the code but part of it, so it cannot be out of date. Types dramatically ease refactoring and maintenance of old code bases

Bad type systems

All programs work

Programs that are well-typed

Zone of Abysmal Pain
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.
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

good expressions

bad expressions

all expressions
The program computes with words; its up to 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

- Float
- Char
- Boolean
- Integer
- Double
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?
Records

Record types allow related data of heterogeneous types to be stored and manipulated together. Originally introduced by COBOL, records also appeared in Algol 68, which called them structures, and introduced the keyword `struct`. Many modern languages, including C and its descendants, employ the Algol 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.
Example Records

type Element is
  record
    Name : String (1..2) := (others => ' ');
    Atomic_Number : Natural;
    Atomic_Weight : Long_Long_Float;
    Metal : Boolean;
  end record;
pragma pack (Element);  -- 33 bytes down to 22 bytes
Example Records

type Element is
  record
    Name : String (1..2) := (others => ' ');
    Atomic_Number : Natural;
    Atomic_Weight : Long_Long_Float;
    Metal : Boolean;
  end record;
pragma pack (Element);    -- 33 bytes down to 22 bytes
Example Records

type Element is
    record
        Name : String (1..2) := (others => ' ');
        Atomic_Number : Natural;
        Atomic_Weight : Long_Long_Float;
        Metal : Boolean;
    end record;
pragma pack (Element);  -- 33 bytes down to 22 bytes
Example Records

```
struct element {
    char name[2];
    int atomic_number;
    long double atomic_weight;  // __float128
    _Bool metal;    / C99 native type
};
```
Example Layout of Record

- Name field: two bytes for the string
- Two bytes of padding for word alignment of integers
- Atom number field: 32 bits, one word, unsigned integer
- Eight bytes, two words padding for double-word alignment for long floating point
- Atom weight field: 16 bytes, four words for 128 bit floating-point number
- Metal field: One byte, smallest storage unit, for boolean value
- Three bytes wasted as nothing will be stored except aligned starting at word or double-word address
Composite Types

<table>
<thead>
<tr>
<th></th>
<th>homogeneous</th>
<th>heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>static</td>
<td></td>
<td>record</td>
</tr>
<tr>
<td>dynamic</td>
<td>array</td>
<td></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
dynamic
foreach $item (%days) {
    print "The number is: $item.\n";
}
```

Often implemented using hash tables
Sebesta 11th, 6.5 Array Types.
Scott, 4th 8.2.2. Dimensions, Bounds, and Allocation and 8.2.2. Memory Layout
Arrays

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>8</td>
</tr>
</tbody>
</table>

Row-major order: 3, 4, 7, 6, 2, 5, 1, 3, 8. Column-major order (uncommon): 3, 6, 1, 4, 2, 3, 7, 5, 8.
A procedure type is needed in programs that pass procedures as arguments. Ada example:

```ada
funarg/trapezoid.adb```

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.

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
   1. Subranges
   2. Variant records
   3. Subprocedures as arguments
3. Russell’s paradox
Pointer to “Anything”

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.
(*) 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 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

- Boolean
- Character: range 'A'..Z'
- Float: digits 4, range 0..125
- String: strings of length 10, strings of length 5
- Integer: range -50..25, range 0..100
- Integer: range 1..500

All other types
Runtime Errors

Some languages do not check for runtime errors. This leads to something worse than a runtime error—the absence of a runtime error. For example, by not checking array indexes or doing illegal operations on pointers, C and C++ allow the program to write over useful parts of memory. This is called a buffer overflow and is the cause of much malware. This is not possible in Java, Ada (checks can be suppressed), C#, Python, Modula-3, Haskell, or any recently designed high-level language.
Runtime Errors

Java was designed so that compiler would detect at compile-time many things which would be runtime errors in other languages. No language can eliminate runtime errors altogether by improved compile-time checking. Some Java runtime errors (exceptions) represent a logical error (a mistake, a bug, by the programmer as opposed to, say, bad luck) in the program.

- `java.lang.ArithmeticException` (e.g., division by zero)
- `java.lang.NullPointerException` (“billion dollar mistake”)
- `java.lang.ArrayIndexOutOfBoundsException`

**Billion Dollar Mistake** — presentation by Hoare in 2009.

Some programming languages have evolved more safeguards, e.g., option 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.

The Morris worm is the only malware that I have personally experienced. Morris is now an MIT professor.
Random Fact 6.1  An Early Internet Worm

In November 1988, Robert Morris, a student at Cornell University, launched a so-called virus program that infected about 6,000 computers connected to the Internet across the United States. Tens of thousands of computer users were unable to read their e-mail or otherwise use their computers. All major universities and many high-tech companies were affected. (The Internet was much smaller then than it is now.)

The particular kind of virus used in this attack is called a worm. The virus program crawled from one computer on the Internet to the next. The worm would attempt to connect to finger, a program in the UNIX operating system for finding information on a user who has an account on a particular computer on the network. Like many programs in UNIX, finger was written in the C language. In C, as in C++, arrays have a fixed size. To store the user name to be looked up (say, walters@cs.sjsu.edu), the finger program allocated an array of 512 characters, under the assumption that nobody would ever provide such a long input. Unfortunately, C, like C++, does not check that an array index is less than the length of the array. If you write into an array using an index that is too large, you simply overwrite memory locations that belong to some other objects. In some versions of the finger program, the programmer had been lazy and had not checked whether the array holding the input characters was large enough to hold the input. So the worm program purposefully filled the 512-character array with 536 bytes. The excess 24 bytes would overwrite a return address, which the attacker knew was stored just after the line buffer. When that function was finished, it didn’t return to its caller but to code supplied by the worm (see Figure 4). That code ran under the same superuser privileges as finger, allowing the worm to gain entry into the remote system. Had the programmer who wrote finger been more conscientious, this particular attack would not be possible. In C++, as in C, all programmers must be very careful not to overrun array boundaries.

One may well speculate what would possess the virus author to spend many weeks to plan the antisocial act of breaking into thousands of computers and disabling them. It appears that the break-in was fully intended by the author, but the disabling of the computers was a bug, caused by continuous reinfection. Morris was sentenced to 3 years probation, 400 hours of community service, and fined $10,000.

In recent years, computer attacks have intensified and the motives have become more sinister. Instead of disabling computers, viruses often steal financial data or use the attacked computers for sending spam e-mail. Sadly, many of these attacks continue to be possible because of poorly written programs that are susceptible to buffer overrun errors.

Figure 4  A “Buffer Overrun” Attack
Runtime Guarantees Better Than Nothing

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 *p) {
    char buf[100];
    char *p = buf;
    do {
        /* read from stream s: */
        *p++ = getc(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.
The whole economic boom in cybersecurity seems largely to be a consequence of poor engineering. We have allowed ourselves to become dependent on an infrastructure with the characteristics of a medieval firetrap—a maze of twisty little streets and passages bordered by buildings highly vulnerable to arson.
To a disturbing extent the kinds of underlying flaws exploited by attackers have not changed very much. . . . One of the most widespread vulnerabilities found recently, the so-called Heartbleed flaw in OpenSSL, was apparently overlooked . . . for more than two years. What was the flaw? Failure to apply adequate bounds-checking to a memory buffer.

Carl Landwehr, CACM, 2015, pages 24–25
In Kotlin, the type system distinguishes between references that can hold null (nullable references) and those that cannot (non-null references).

```kotlin
// Regular initialization means non-null by default
var a: String = "abc"
a = null // compilation error

var b: String? = "abc" // can be set null
b = null // ok
```
var a: String = "abc"
var b: String? = "abc"

val l1 = a.length // Save the length in an Int
val l2 = b.length // error: variable 'b' can be null
val l3 = if (b != null) b.length else -1
val l4 = b?.length // Int? value might be null
The hair style of Elvis Presley was well-known

Elvis operator ?: in Kotlin

```kotlin
var b: String? = "abc"
val l = b?.length ?: -1
```
What are variant records?
Out of fashion because of subclasses in OO languages.

- Euclid – type safe
- Ada – runtime checking

variant/vr.adb
Two interpretation of the same bits.

(* 22222222222222222222222222222222222222 *)

```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
```
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.

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

\[
\text{program Main (input, output);} \\
\text{procedure Print (x: integer; \ procedure } P); \\
\text{begin} \\
\text{if } x=1 \text{ then } P (2.1) \text{ else } P (3.2, 9.3); \\
\text{end \{ Print \};} \\
\text{procedure Print1 (x: real); ...} \\
\text{procedure Print2 (x,y: real); ...} \\
\text{begin \{ Main \}} \\
\text{Print (1, Print1); \ Print (2, Print2);} \\
\text{end.}
\]
Procedure arguments declared with argument types.

\begin{verbatim}
P (x:integer; procedure F(a:real;b:char); y:real);
P (x:integer; function F(c:real):real; y:real);
\end{verbatim}

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.
Russell’s Paradox

The absence of type checking not only creates the possibility that the bits will be misinterpreted, but also undermines the foundations of programming. It is a manifestation of the halting problem and the limits of computation. A programming language ought to prevent logical nonsense that leads to non-terminating programs.
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.
Scott 4th, Section 7.2.1, Type Equivalence.
Tucker & Noonan 2nd, Section 5.7 Type Equivalence.
Watt, Section 2.5.2, Type Equivalence.
Wilson & Clark 3rd, Name and structural equivalence, page 165–166.
Structural equivalence requires a clever algorithm

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

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

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

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

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

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

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

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

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

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

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

```plaintext
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

```plaintext
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;
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.

```plc
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 type
  -- 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).

```ada
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.

```plaintext
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;

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 or synonyms) for the same type. Perhaps because an “official” name is too long. Is it possible to get a type synonym in Ada?

```
type T is Integer;  -- synonym? No, it’s a new type.
```

Ada type declarations are generative. Contrast this with Haskell

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

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

```
subtype T is Stacks_And_Queue.Stacks.Internal_Representation;
```
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 are generative; they 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.

```apl
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.

```plc
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
  N := K;  -- illegal
  J := K;  -- illegal

  N := N+J;  -- legal
  K := K+J;  -- illegal
  K := K+1;  -- legal
  J := J+1;  -- legal
end;
```
Name equivalence does not provide abstraction (hence the need for the abstraction mechanisms to be discussed later under the topic of modules).

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;
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></td>
<td></td>
<td>Inclusion aka Subtype</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:

- `real: int -> float`
- `floor: float -> int`
- `ceil: float -> int`
- `round: float -> int`
- `itoa: int -> string`
- `ord: char -> int`
- `char: int -> char`

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*. 
Implicit Coercions in C

```c
short a = 2000; // sizeof(short)==3
int b;         // sizeof(int)==4
b = a           // guaranteed to work
b = 0xAAAAAAA; // too big for short
a = b;          // implementation dependent narrowing
```
Implicit Coercions in C++

User-defined implicit type coercions in C++

cocerion/tiny.cc
Coercions

Pre-defined implicit type coercions in C++

\[ \ll : \text{istream} * \text{int} \rightarrow \text{istream} \]
\[ \ll : \text{istream} * \text{float} \rightarrow \text{istream} \]

\[
cin \ll i \ll f
\]
\[
\text{while (cin} \ll i)\]
\[
\text{istream to boolean.}
\]
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 this 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 Aaron 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 in C++

C++ can overload operators. From Scott 4, Example 3.26, page 148–149.

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

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

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

```cpp
z = x + y;  // user-defined operator+
z = x.operator+ (y);
```
Overloading

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

```cpp
class X {
   /* ... */
   X& operator=(const X& from) { /* ... */ } // assignment
   Y& operator[](int) { /* ... */ } // x[]
   Z& operator+(int) { /* ... */ } // x+3
   Z& operator+() { /* ... */ } // +x (unary)
};
```

Java decided not to permit overloading of operators.
Cardelli-Wegner Taxonomy (Recap)

<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 aka Subtype</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Parametric Polymorphism

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 package or function (also the keyword new in Ada).
Ada Generics

generic
    type T is private;
function Id (X : in T) return T;

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

-- generic instantiations
function Int_Id is new Id (T => Integer);
function Float_Id is new Id (T => Float);

-- example uses
if (Int_Id (X => 3) = 3) then
    X := Float_Id (X => 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, Data_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++)
}
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>)
```
Universal, Parameteric Polymorphism

Many of the data structures important in computing concern collections: lists, dictionaries, tree, graphs, etc.
The implementation techniques are all about infrastructure support (pointers usually)—what we might call “bookkeeping.”
The importance of the collections are the items in the collection —what we might call the “payload.”
These structures are useful in so many application because the type of the items is immaterial in implementing the structures.
Data structures already implemented in the standard Java library ready to be used on whatever data type is needed.

```java
List<String> index = new ArrayList<>();
List<Person> roster = new ArrayList<>();
List<Point> perimeter = new LinkedList<>();
Map<String, List<String>> graph = new HashMap<>();
```

The instance of the type is inferred from the declaration; same code for all instances of a generic class.
Universal, Parameteric Polymorphism

head [1,2,3,4] --&gt; 1  head::[int] --&gt; int  
head "list" --&gt; 'l'  head::[char] --&gt; char

No declarations; instances are inferred from their use; same code for all instances.
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 point operation.

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 \( \tau \)
Organize all the types into a partial order $\sigma \leq \tau$ such that the implementation will never care that an element of type $\sigma$ is provided when expecting a $\tau$. The set of types (actually a bounded 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.
Make Everything Unrelated Is Easy

```
T
\_/\
\  /  \\
int \_ \_ float
```

T

---

PLC: Types

© 14 October 2022 129 / 190
The solution comes in a surprising place: records.

Example records:

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

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

It is safe to ignore fields you do not know about. It is easy for an implementation to access data values with extra fields tacked on the end.
Inclusion Polymorphism

{c:int}

{a:int, c:int}

{a:int, b:char, c:int}
Consider a subprocedure with a record type as the domain. The implementation of the subprocedure may very well work without modification when given a record with additional fields.

Since the subprocedure does not know about the additional fields of the record given as an actual argument, so it cannot compute with them in any way. The unknown, additional fields are ignored.
Subtype Polymorphism Structure

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

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

\[
\begin{align*}
\{b: \text{char}\} & \quad \{a: \text{int}, b: \text{char}\} \\
\{b: \text{char}, c: \text{int}\} & \quad \{a: \text{int}, b: \text{char}, c: \text{int}\}
\end{align*}
\]
Nominal Subtyping

*Type systems like Java’s, in which names are significant and subtypes are explicitly declared, are called nominal. Type systems in which names are inessential and subtyping is define directly on the structures of types are called structural.*

Pierce, TAPLS, page 252.
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 in Ada

objects/shape_main.adb
Nominal Versus Structural Subtyping

Just for fun, the partial order relation can be easily programmed for both nominal and structural subtyping.
Nominal Subtyping

-- The important part of a class definition is its *name*
-- and its superclass name.
data Type = Bottom | Class{name::String,parent::Type} | Top

instance Bounded Type where
  minBound = Bottom
  maxBound = Top
Nominal Subtyping

-- The subtype relation, a
-- reflexive, anti-symmetric, and transitive (partial order).

(<?,) :: Type -> Type -> Bool

Bottom <: _ = True -- Bottom is <: anything

Class _ _ <: Bottom = False

Class n t <: s@(Class m _) |
| n==m = True -- Same name iff same class
| otherwise = t <: s -- Is parent a subtype?

Class _ _ <: Top = True

Top <: Top = True

Top <: _ = False -- Top is not <: anything (except Top)
The important part of a record is its structure: the list (set really) of fields and their types.

data Type = Bottom | Struct [(String, Type)] | Top

instance Bounded Type where
  minBound = Bottom
  maxBound = Top
-- The subtype relation is defined indirectly
(\<:\) :: Type -> Type -> Bool
t <: s = (lub t s) == s

lub :: Type -> Type -> Type
lub Bottom s = s
lub t Bottom = t

-- Data.List.intersect :: Eq a => [a] -> [a] -> [a]
lub (Struct t) (Struct s) = Struct (intersect t s)
lub Top _ = Top
lub _ Top = Top
Type Constructors

Consider this list of types:

- Bool
- Integer
- Maybe
- Array

It is obvious what a Bool or what an Integer is: it describes some value at runtime, such as True or 42. The same cannot be said about Maybe, or Array. There are no values of type Array, that makes no sense. Array is not a proper type of its own, but rather a *type constructor*. It is used, for example, to construct a type for array of integers out of the type for integers.

Function types are similar; they construct a type for every domain and range.
Unary Type Constructors

When does the following hold?

array of $\sigma \ <: \ array \ of \ \tau$

Three possibilities:

1. $\sigma \leq \tau$ co-variant
2. $\tau \leq \sigma$ contra-variant
3. $\sigma = \tau$ neither
Unary Type Constructors

When does the following hold?

option of $\sigma <: \text{option of } \tau$

Three possibilities:

1. $\sigma \leq \tau$ co-variant
2. $\tau \leq \sigma$ contra-variant
3. $\sigma = \tau$ neither
Subtyping of Array Types

Records, OK. How about variant records?

Variant records are the dual of records.

Records and variant records, OK. How about arrays?

There are three possibilities for a unary type constructor. Are arrays co-variant, contra-variant, or neither?
Read-only data types (sources) can be covariant; write-only data types (sinks) can be contravariant. Mutable data types which act as both sources and sinks should be invariant.

To illustrate this consider the array type – a mutable data type. For the type Animal we can make the type Animal[], which is an "array of animals".

```java
class Animal {}
class Cat extends Animal {}
class Dog extends Animal {}
```
We have the option to treat this as either:

- **covariant**: a `Cat[]` is an `Animal[]`;
- **contravariant**: an `Animal[]` is a `Cat[]`;
- **invariant**: an `Animal[]` is not a `Cat[]` and a `Cat[]` is not an `Animal[]`.

If we wish to avoid type errors, then only the third choice is safe.
Clearly, not every Animal[] can be treated as if it were a Cat[], since a client reading from the array will expect a Cat, but an Animal[] may contain, e.g., a Dog. So the contravariant rule is not safe, to wit:

```java
Animal[] animals =
    new Animal { new Cat(), new Dog(), new Animal() };

client (animals)

tvoid client (Cat[] cats) {
    // cats[1] is a cat? No, it’s a dog!
}
```
Conversely, a Cat[] cannot be treated as an Animal[]. It should always be possible to put a Dog into an Animal[]. With covariant arrays this cannot be guaranteed to be safe, since the backing store might actually be an array of cats. So the covariant rule is also not safe—the array constructor should be invariant.

```java
Cat[] cats =
    new Cat {new Cat(), new Cat(), new new Cat()};

client (cats)

void client (Animals[] animals) {
    cats[0] := new Dog();
}

// cat[0] is a cat?, No, it’s a dog!
```
Note that this is only an issue for mutable arrays; the covariant rule is safe for immutable (read-only) arrays. Likewise, the contravariant rule would be safe for write-only arrays.
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.)

misc/Sub.java – array
The type of a function $\tau = \tau_d \to \tau_r$ has a set of values of the domain $\tau_d$ and the range $\tau_r$. When is the type $\sigma$ a subtype of $\tau$? That is, when is $\sigma \leq \tau$? Clearly $\sigma$ must be the type of a function, so, when is $\sigma_d \to \sigma_r \leq \tau_d \to \tau_r$?
Subtyping of Function Types

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

When you apply the function you get a value in its range $\sigma_r$. If you were prepared for all values in $\tau_r$, then all is well.
Subtyping of Function Types

When you apply the function you get a value in its range $\sigma_r$. If you were prepared for all values in $\tau_r$, then all is well.

If you apply the functions only to values in $\tau_d$, but the function accepts all values in $\sigma_d$, then all is well.
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 \( \tau_d \rightarrow \tau_r \leq \sigma_d \rightarrow \sigma_r \).

If \( \sigma_d \leq \tau_d \) and \( \tau_r \leq \sigma_r \), 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.

Now Java has lambdas.
class Mammal {}
class Bat extends Mammal {}
class Vampire extends Bat {}

class Food {}
class Vegetable extends Food {}
class Bamboo extends Vegetable {}

// In java.lang.function
public interface Function<T,R> {
    R apply(T t);
}
Subtype Polymorphism

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

Functions are contravariant in the domain and covariant in the range.
Now we interject overloading and dynamic dispatch, and contrast subtype polymorphism with parametric polymorphism.

(Could talk about OO programming later, but it is not worth it.)

Later we talk about combining subtype and parametric polymorphism.

Finally, “duck” typing.
Subtype Polymorphism and OO

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

objects/inheritance.adb
objects/dynamic.adb
objects/overload.adb
class/Contra.java – covariance in the range
class/Vegetarian.java
Object Oriented Programming

• Sebesta 11th. Chapter 12: Support for Object-Oriented Programming
  • Section 12.5: Implementation of Object-Oriented Constructs
    Dynamic dispatch implemented using a virtual method table.
With static method binding (as in Simula, C++, C#, or Ada 95), the compiler can always tell which version of a method to call, based on the type of the variable being used. With dynamic method binding, however, the object referred to by a reference or pointer variable must contain sufficient information to allow the code generated by the compiler to find the right version of the method at run time. The most common implementation represents each object with a record whose first field contains the address of a virtual method table (vtable) for the object’s class.
A hybrid form of polymorphism is found in recent versions of the programming language Java and C#. It is a combination of parametric polymorphism and subtype polymorphism known as bounded quantification polymorphism.
We motivate the need for bounded quantification by a series of examples in a language of records, variant records, and functions similar to that of Cardelli’s Amber language.

- record types are written with curly braces \{a:int, b:char\}
- record are written with curly braces \{a=3, b='c'\}
- (variant record types are written with square brackets \[a:int, b:char\])
- (variant record are written with square brackets \[a=3\])
- function declarations are written like this \texttt{fun f(x) = x+1} with value parameters written in parentheses.
- function types are written with an arrow like this \texttt{int->int}
fun f (x:{a:int}) = x; (* f:{a:int}->{a:int} *)

val ra = {a=0} (* record with "a" field *)

f (ra); (* applied to same type *)
{a=0} : {a:int}

val rab = {a=0,b=true} (* record with "a", "b" field *)

f (rab); (* subtype polymorphism strips the type *)
{a=0,b=true} : {a:int}
The return value as “laundered” — stripped of the information that it contains a b field.
Think of Java programming before the introduction of parametric polymorphism.
Define a new function \( \text{fpoly} \) with a type parameter and a value parameter. Parametric polymorphism does not strip the field information from the return type. Type parameters are written in square brackets, begin with an apostrophe and a uppercase letter.

\[ (* \text{parametric polymorphic version \( \text{fpoly}:'X\rightarrow'X \) } *) \]
\[
\text{fun \( \text{fpoly} ['X] (x:'X) = x; \)}
\]

\[ (* \text{applied to a record; type is ok } *) \]
\[
\text{fpoly } [{a:int,b:bool}] \text{ rab; }
\]
\[
\{a=0,b=true\} : \{a:int,b:bool\}
\]

When we pass a record type to \( \text{fpoly} \) we can apply it to a record. Notice the knowledge of the \( b \) field is not lost.
Take a more complicated function and we want subtype polymorphism, but without “stripping” the knowledge of the records fields.

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

```haskell
f2 ra;
{orig={a=0}, s=1} : {orig:{a:int}, s:int}
```

Works fine on an record with one field.
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".

How do we preserve the type and require presence of a field? Neither subtype polymorphism nor parametric polymorphism works.
The solution is a combination of parametric and subtype polymorphism.

\[
f2poly [\textbf{'X} <: \{a:\text{int}\}] (x:\textbf{'X}) = \{\text{orig}=x, s=x.a+1\};
\]

\[
f2poly : \textbf{'X}<:{a:\text{int}} \rightarrow \{\text{orig}:'\textbf{X}, s:\text{int}\}
\]

\[
f2poly \ rab
\]

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

Pierce, Chapter 26 Bounded Quantification.
Polymorphism in Java

• polymorphism/Example2.java
Polymorphism in Java

- polymorphism/Coercion.java – subtype polymorphism
- polymorphism/Overloading.java – universal, parametric polymorphism
- polymorphism/Shapes.java – universal, parametric polymorphism
- polymorphism/Parametric.java – universal, parametric polymorphism
- polymorphism/Wild.java – bounded polymorphism
- polymorphism/Move.java – bounded polymorphism
- polymorphism/Example.java – bounded polymorphism
Parameteric Polymorphism in Java

Java generics reference?
Java type erasure ensure generics incur no runtime overhead. Reference?
C++ templates reference?
Contravariance and Overloading

Even though this is safe for programming languages to do, Kotlin and Scala have followed Java’s lead, and do not allow for contravariance on argument types in normal class and interface inheritance, in order to allow for method overloading.
Wild Card

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-Supplies*
Java wildcard cube

See Wikipedia Co and Contra

© 14 October 2022
Implicit nulls were invented by Hoare, who refers to this invention as his billion-dollar mistake. The feature has been a cause of many bugs in software. It adds a case that is easy to forget and difficult to keep track of and reason about. Here it causes the same problem for the same reasons, but at the type level. The reasoning for wildcards and path-dependent types would be perfectly valid if not for implicit null values.

Amin & Tate, Java and Scala’s Type Systems are Unsound: The Existential Crisis of Null Pointers, OOPSLA 2016.
Kotlin

Kotlin does not have Java’s wild cards. It uses `in` and `out` on types. (This is not to be confused the value parameter intent, see Fortran or Ada.) See [Generics](#) in the Kotlin reference documentation. See this blog post [here].
From the movie *Coconuts*, 1929

*Chico*: Alright, why a duck?

*Hammer [Groucho]*: [Pause] I’m not playing “Ask Me Another,” I say that’s a viaduct.

*Chico*: Alright! Why a duck? Why that ... why a duck? Why a no chicken?

*Hammer [Groucho]*: Well, I don’t know why a no chicken; I’m a stranger here myself. All I know is that it’s a viaduct. You try to cross over there a chicken and you’ll find out why a duck.
class Duck:
    def __init__(self, n, w="waddle"):  
        self.name = n
        self.walk = w

    def talk(self):
        return self.name + " : quack! quack!"

harpo = Duck()
del harpo.walk

# "Monkey patching" =:)
harpo.talk =
    type.MethodType(lambda self: "honk! honk!", harpo)
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

See this stackoverflow question and answer about reflection: What and Why Reflection

See this stackoverflow question and answer about monkey patching.
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