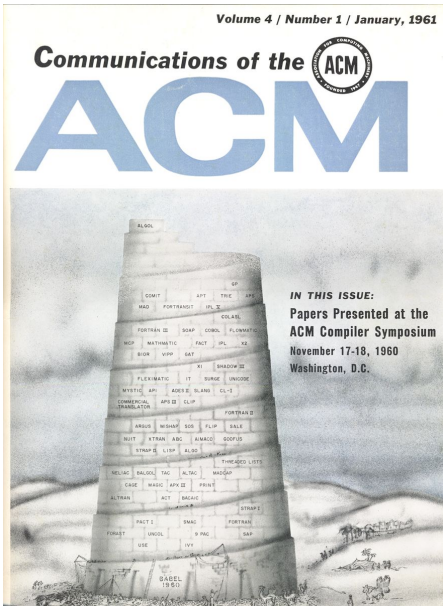
Programming Languages

CSE 3302, Summer 2019
Module 07
Type Systems



M07

Type Systems

Type Systems Intro …

- Computer memory is naturally *untyped*.†
  - Bits in memory are just that—*bits*. By themselves they have neither structure nor interpretation.
  - They have meaning only in how they are *interpreted* and *operated upon* by instructions.

- A *Type System* is a way to bring structure and interpretation to those bits.
  - The bits are then seen as an *encoding* of a particular *value*.

†There are some notable exceptions to this general rule (e.g., the Symbolics Lisp Machine), but these exceptions are few and far between so we will tacitly skip over them here.

[*Encodings*]

- The 32 bits `01000110010100100100010101000100` *could* mean …
  - The `unsigned int` `0x46524544`.
  - The `int` `1,179,796,804`.
  - The `float` `13,457.316406`.
  - The `char`s `FRED`.
  - …
- Without interpretation imposed by a type or an operation, no one answer is correct.

Type Systems Intro …

- Everyone has an *intuitive* notion of what types are.
- There are *three* popular ways to *formalize* the notion of type.
  - Though, of course, there are *mixtures* of these views.

① A *shorthand name for a collection of values* (a *domain*).

- E.g., the *8-bit signed integers*.
- Could be more complex, though, e.g., the *functions that take one* `int` *argument and return a* `double` *result*.
- Known as the *Denotational* view.

Type Systems Intro …

② Something built out of (smaller) components.

- `int`, `real`, `string`, … The *fundamental* types.
- `record`, `tuple`, `map`, … The *composite* types.
- Known as the *constructive* or *structural* view.

③ A collection of *well-defined operations*.

- A type is what it *does* (or what can be *done to it*); an *interface*.
- Known as the *abstraction* or *object-oriented* view.

Type Systems Intro …

- Types are associated with *expressions* and *objects*.
- The type of an *expression* might *not* be the type of any of the values that occur in it! ①
- (An *object* here is anything that can have a name.)

```
Python 3.6.3                                          ①
(default, Oct 3 2017, 21:45:48)
[GCC 7.2.0] on linux
>>> type( 1 )
<class 'int'>
>>> type( 2 )
<class 'int'>
>>> type( 1 / 2 )
<class 'float'>
```

Type Systems Intro …

- What has a type?
  - Named or Literal constants, Variables, Expressions, Record fields, Parameters, Functions, Subroutines (sometimes), Objects, …
  - Depends on the language.
- A *name* might have one type associated with it (think variable declaration), but at times might refer to an object of a *different* type.

[*Declared vs. Actual Type*]

- Example,

```
Person p = new Student( "John Doe" )
```

- The variable `p` is *declared* to be type `Person` and that is its *apparent* type.
- The *apparent* type is *static*. It's associated with `p` at compile time via `p`'s declaration.

[*Declared vs. Actual Type*]

- Example,

```
Person p = new Student( "John Doe" )
```

- However, the type of the value of the object `p` is holding — its *actual* type — is `Student`.
- The *actual* type is *dynamic* since it can change at run time.
  - At one point, `p` might be holding a `Student`. At another, `p` might be holding a `Person`. And so on …

[*Declared vs. Actual Type*]

- Example,

```
Person p = new Student( "John Doe" )
```

- The *declared* type controls *which* methods can be called.
  - Since `p` is of *declared* type `Person`, only methods defined for `Person` can be called for `p`. (The compiler enforces this.)
  - Even if `Student` has additional methods, they cannot be called for `p`.

[*Declared vs. Actual Type*]

- Example,

```
Person p = new Student( "John Doe" )
```

- The *actual* type controls which *implementation* of the method is called.
  - Since `p` is of *actual* type `Student`, `Student`'s implementation of a method would be called.
  - This is *dynamic method dispatch* and is determined at *run time*.

[*Declared vs. Actual Type Example*]

The Java sample code to the right gives the results shown below.

Notice how even though `refP` *is declared as a* `Person`*, it can hold a reference to a* `Student` *and* `Student`*'s* `f()` *method is invoked. This is an example of* dynamic method dispatch.

On the other hand, even though methods *are dynamically dispatched,* member variables *are* not. *Even when the reference is to a* `Student`*, the underlying* `Person` *object's* `who` *member is printed.*

```
--- refP is to a 'Person'
Person.f
Person
--- refP is to a 'Student'
Student.f
Person
--- refS is to a 'Student'
Student.f
Student
```

Java

```
class Person {
 String who = "Person";
 void f() { System.out.println( "Person.f" ); } }

class Student extends Person {
 String who = "Student";
 void f() { System.out.println( "Student.f" ); } }

class DynamicMethodDispatch {
 public static void main( String args[] ) {
   Person refP;
   Student refS;

   System.out.println( "--- refP is to a 'Person'" );
   refP = new Person();
   refP.f();
   System.out.println( refP.who );

   System.out.println( "--- refP is to a 'Student'" );
   refP = new Student();
   refP.f();
   System.out.println( refP.who );

   System.out.println( "--- refS is to a 'Student'" );
   refS = (Student) refP;
   refS.f();
   System.out.println( refS.who );
 }
}
```

[*Declared vs. Actual Type Example*]

The C# version of the sample code gets the same answers as the Java version.

On the other hand, the C# compiler gives an informative warning message about the shadowing:

```
warning CS0108: `Student.who' hides
inherited member `Person.who'. Use the
new keyword if hiding was intended
```

```
--- refP is to a 'Person'
Person.f
Person
--- refP is to a 'Student'
Student.f
Person
--- refS is to a 'Student'
Student.f
Student
```

C#

```
class Person {
  public String who = "Person";
  public virtual void f() { Console.WriteLine( "Person.f" ); } }

class Student : Person {
  public String who = "Student";
  public override void f() { Console.WriteLine( "Student.f" ); }
}

class DynamicMethodDispatch {
  public static void Main( String [] args ) {
    Person refP;
    Student refS;

    Console.WriteLine( "--- refP is to a 'Person'" );
    refP = new Person();
    refP.f();
    Console.WriteLine( refP.who );

    Console.WriteLine( "--- refP is to a 'Student'" );
    refP = new Student();
    refP.f();
    Console.WriteLine( refP.who );

    Console.WriteLine( "--- refS is to a 'Student'" );
    refS = (Student) refP;
    refS.f();
    Console.WriteLine( refS.who );
  }
}
```

[*Declared vs. Actual Type Example*]

If we use a method *to retrieve the value of the* who *member variable, we will get the proper result.*

The `getWho()` *method returns the value of the* who *member variable of its own instance. Which* `getWho()` *gets run is selected by* dynamic method dispatch.

```
--- refP is to a 'Student'
Student.f
.who returns      : Person
.getWho() returns : Student
```

Java

```java
class Person {
 String who = "Person";
 void f() { System.out.println( "Person.f" ); }
 String getWho() { return who; } }

class Student extends Person {
 String who = "Student";
 void f() { System.out.println( "Student.f" ); }
 String getWho() { return who; } }

class DynamicMethodDispatch {
 public static void main( String args[] ) {
   Person refP;

   System.out.println( "--- refP is to a 'Student'" );
   refP = new Student();
   refP.f();
   System.out.println( ".who returns      : " + refP.who );
   System.out.println( ".getWho() returns : " + refP.getWho() );
 }
}
```

Type Systems Intro …

- Types may be *built-in* (e.g., C's `int`, `double`, `char`) or *user-defined* (e.g., with C's `enum`, `typedef`, Python's `class`).
- Types may be *explicitly declared* (C ①, C++, Java, Ada, …) or *implicitly inferred* (ML, OCaml ②, Scheme, Python, Haskell, …).

①

```c
int fibonacci( int n ) {
 return fibTR( n, 0, 1, 0);
}

int fibTR( int n, int n1, int n2, int i ) {
 return i == n ? n2
        : fibTR( n, n2, n1+n2, i+1 );
}
```

②

```
let fibonacci n =
 let rec fibTR n1 n2 i =
   if i = n then n2
   else fibTR n2 (n1 + n2) (i + 1) in
 fibTR 0 1 0;;
```

Type Systems Intro …

- Types provide *implicit context* for many operations so the programmer does not have to do so explicitly.
  - `a + b` ⇒ the types of `a`, `b` determine what `+` means.
  - Summation for integers, concatenation for strings.

- Types limit the *set of operations* that may be performed.
  - For example, can't add a `double` and a `union`.
  - Usually can add a `double` and an `int`, but only after coercion.
  - Not perfect error catcher, but right often enough to be useful.

Type Systems Intro …

- If specified explicitly, types often can make the source program *easier to read* and *easier to understand*.
  - They are a kind of *stylized* documentation with compiler-enforced *correctness*.
  - On the other hand, can make the program *harder to write*.

- If known at compile time, types can enhance *performance optimization*.
  - Operation selection, some kinds of alias avoidance, etc.

What's a Type System?

- A *Type System* consists of …
  - A *mechanism* to *define types* and to *associate* them with language constructs.
  - A set of rules for
    - *Type Equivalence* — When are the types of two values the same?
    - *Type Compatibility* — When can a value of a given type be used in a given context?
    - *Type Inference* — What is the type of an expression given the types of its constituents and (sometimes) the context?

Type Checking

- *Type Checking* is the process of ensuring that the *type compatibility* rules are being followed.
  - When they aren't, a *type clash* is said to have occurred.
- A *strongly-typed* language is one that prohibits *in an enforceable way* the use of an operation on an object that does not support it.
  - Otherwise it is *weakly typed* or even *untyped* (e.g., assembler).

Type Checking

- If this enforcement is done at *compile time*, the language is said to be *statically typed*.

- Static Type Checking results in more *efficient* code.
  - The type of every node in the AST (Abstract Syntax Tree) is checked at compile time.
  - Some level of *type inference* is always necessary.
  - Often, *type declarations* are used to minimize the required inference steps.

Type Checking

- If the enforcement is at *run time*, the language is said to be *dynamically typed*.

- Dynamic Type Checking is *flexible*.
  - Types are checked as operations are carried out.
  - Every time an object is accessed (a variable, argument, function, …), its type is checked for compatibility in the current context.

Type Checking

- There's a set of tradeoffs among the strength of the type checking and when the checks are made.
- *Strong* and *Static* results in verbose source code (all those declarations) but the checks are done pre-runtime for efficiency.
- *Strong* and *Dynamic* involves less coding but one doesn't find out about the errors until run time.
- *Weak* and *Dynamic* requires the least coding but one doesn't find out about errors until run time, if *even then*!
  - Results can be "approximate" because of the weak typing.

Type Safety

- Aside from providing information to the user and the compiler, types help ensure data is used correctly.
- *Type Safety* is enforced by the compiler (or run-time system) to ensure data is not used in an *incorrect* or *meaningless* way.
- A language without type safety is highly prone to *errors* and *exploits*.
  - Almost all (modern) languages support type safety at least to some extent.
  - Type safety checks can be overridden in some languages.

Type Checking Examples

- Java is *strongly typed* with type checks at *both* compile time and run time.
  - As much as possible is checked statically, e.g., ①
  - Some checks *have* to be at run time, e.g., casting ②, dynamically loaded libraries.

①

```
String a = 1;   // Static error

Double d = 1.0;
int i = d;      // Static error
```

②

```
Object o = "Not an int";

// Dynamic error
Integer j = (Integer) o;
```

Type Checking Examples

- Python is *strongly typed* with type checks at run time. ①
- Perl is *weakly typed* with type checks at runtime. ②
  - `$a + $b` result seems reasonable.
  - Maybe got lucky there.
  - `$c + $a` result seems ridiculous.

①

```
a = 1
b = "2"

// Dynamic error
print( a + b )
```

②

```
$a = 1;
$b = "2";
$c = "q";

# No error; prints 3
print $a + $b;

# No error; prints 1
print $c + $a;
```

Type Checking

- Most languages are *strongly typed*, but there is a spread over when the type checks occur.
  - In Ada and Pascal type checks are at compile time.
  - Java type checks occur at both compile and run time.
  - For Common Lisp, Python, and Ruby, the type checks are at run time.

[*Is C Strongly-Typed?*]

- C requires that all variables be declared with a specific type.
- This is checked at compile time.
- *Specific types* and *checked at compile time* so *statically typed*, right?
- Not so fast!
- C *also* allows atrocious *type-punning*!
  - `f = *(float *) &ui;`
- The code ① prints `0x3F000000, 0.500000`
- The *bits* of `ui` have been interpreted as a `float` even though it is an `unsigned int`.

①

```c
#include <stdio.h>

int main( int argc, char *argv[] )
{
  unsigned int ui = 0x3F << 24;
  float f;

  f = *(float *) &ui;
  printf( "0x%X, %f\n", ui, f );
}
```

[*Is C Strongly-Typed?*]

- The types are certainly not *enforced*.
  - C does no dynamic checking whatsoever so that "loophole" breaks type.
- It comes down to *definitions*.
- Since there no *enforcement*, C is *by our definition* ***not*** strongly typed.
- (If one stays away from type punning however …)

Dennis Ritchie, creator of C and co-creator of UNIX, has called C "strongly typed and weakly checked".

"To this day, many C programmers believe that *strong typing* just means *pounding extra hard on the keyboard*." —Peter van der Linden, author of *Expert C Programming*.

Polymorphism Overview

- *Polymorphism* applies to *code* (data structures *and* routines) that is designed to work with values of multiple types.
- The types have characteristics in common and the code relies on only those.
- *Parametric Polymorphism* — The code takes a (set of) type(s) as a parameter, either explicitly or implicitly.
- *Subtype Polymorphism* — The code is designed to work with type T, but also works with *refinements* or *extensions* of T.

Polymorphism Overview

- *Duck Typing* — An object is *assumed* to be of an appropriate type as long as it supports whatever method is being invoked.
  - Requires a *dynamic semantic check* to ensure the method is supported.
  - An *operational* style of checking type compatibility.
  - "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."

James Whitcomb Riley
Writer, Poet, Best-selling Author
1849-1916

Orthogonality

- *Orthogonality* is the lessening of restrictions on the ways in which features can be combined.
  - Features can be combined arbitrarily and produce consistent results.
  - An *orthogonal* language is *easier to understand*, *to use*, and *to reason about formally*.
- Having a more *orthogonal* type system in a language is a useful design goal.
- Example, Pascal is more orthogonal than Fortran in that it allows arrays of *any* element type. It is not *completely* orthogonal as it prohibits, e.g., variant records as arbitrary fields of other records.

Classification of Types

- The terminology of types varies from language to language.
- Most languages provide built-in types representing those supported by common processors.
  - *Integers*, *Characters*, *Booleans*, *Reals* (*Floating Points*), …
  - *Characters* started off as single bytes (ASCII, EBCDIC, …), now two or four bytes (UNICODE).
  - *Booleans* are typically one byte, but some languages will pack them into single bits.

Numeric Types

- Many languages have *Integers* and *Reals* and leave it at that.
  - Notation varies!
- A lack of *specified precision* can lead to *portability* problems.
  - Different implementation may use different precisions and thereby get *different* answers from the same program.
- C and C++ distinguish *lengths* of integer and real numbers (e.g., `char`, `short`, `int`, `long`, `float`, `double`).
- The *precisions* are in order, but no actual size is guaranteed.†
  - sizeof(`char`) ≤ sizeof(`short`) ≤ sizeof(`int`) ≤ sizeof(`long`)

†The C and C++ standards *do* guarantee *minimum* sizes, but not *absolute* sizes.

Numeric Types

- C# and Java go so far as providing different lengths *and specified precision* (how many bits).
  - One can still get overflows, but at least you can control *when* the overflows will happen in a portable fashion.
- Lisp, Scheme, Python avoid the precision problem entirely by providing *arbitrary precision* integers as well as *rationals* (having integer *numerator* and *denominator*) for arbitrary precision *fractions*.

Numeric Types — Miscellaneous

- *Unsigned* integer types are provided by C, C++, C#, Modula 2 (which calls them *cardinals*).
- *Complex* values are supported by C, Fortran, Lisp, and Scheme as built-in types.
  - Other languages (e.g., C++) support complex as a standard library class.
- Ada supports *fixed-point* integers.
  - Represented as an integer with an implied decimal point.
- Several languages support a *binary-coded decimal* type.
  - Numbers are kept as base 10 digits. Avoids round-off problems in financial calculations.

Type Systems Overview

- *Integers*, *Booleans*, and *Characters* are known as *Discrete* types.
  - Also known as *Ordinals*.
- Their domains are *Countable* and each value has a well-defined *Predecessor* and *Successor*.
  - Well, not the first and last values, obviously.
- Two more kinds of discrete types are *enumerations* and *subranges*.

Enumerations

- An *Enumeration* is a set of *Named Values*.
- Example:

```
type weekday = ( sun, mon, tue, wed, thu, fri, sat );
```

- Ordered, so can check *before* / *after*.
- Usable as array index.
- Usable as range for loop variable. E.g.,

```
for today := mon to fri do begin …
```

- *Not* interchangeable with integers!
- Introduced by Niklaus Wirth in Pascal.

Niklaus Wirth — designer of Algol W, Euler, Pascal, Modula, Modula-2, Oberon, Oberon-2, Oberon-07, … Earned Turing Award 1984

Enumerations

- In C on the other hand, an enumeration is purely a syntactic convenience.
- The definition in ① has the same effect as that of ②.
- There is no distinction between an `int` and a `weekday` from a type point of view.

```
enum weekday {                    ①
 sun, mon, tue, wed,
 thu, fri, sat };
```

```
typedef int weekday;              ②
const weekday
 sun=0, mon=1, tue=2, wed=3,
 thu=4, fri=5, sat=6;
```

Enumerations

- Typically represented as a small integer, usually a consecutive range starting at 0.
- These *ordinal* values can be significant.
  - Typically *ordinal* and *position* functions are provided to convert to / from the enumeration type and the underlying integer.
  - Ada calls them `pos()` and `val()`. Other languages use `pos()` and `ord()`.

Enumerations

- Several languages allow the specification of the enumeration's integer values.
- For example, C, C++, C# allow ①.
- Ada expresses the same concept as ②.

① C, C++, C#

```
enum armRegisters = {
 fp=7, sp=13, lr=14, pc=15
};
```

② Ada

```
type armRegisters is (
 fp, sp, lr, pc
);

for armRegisters use (
 fp=>7, sp=>13, lr=>14, pc=>15
);
```

Enumerations

- Pascal and C do not allow the same name to be used in multiple enumerations.
- Java and C# allow this, but require fully-qualified names.
- Ada allows this and will complain only if the compiler cannot infer the type of the name from the context.
  - We saw this in the *Overloading* example in *M04*.
- C++ traditionally was the same as C but C++11 relaxed the requirement and is now similar to Java and C#.

Subranges

- Also introduced in Pascal, a *Subrange* type is a *Contiguous Subset* of the values of some discrete *Base* (or *Parent*) type.
- For example, in Ada one can declare

```
type testScore is new integer range 0 .. 100;
subtype workday is weekday range mon .. fri;
```

- `testScore` is a *Derived* type and incompatible with `integer`.
- `workday` is a *Constrained* subtype and can be intermixed with `weekday` values.

Subranges

- Subranges are useful for *correctness*, *documentary*, and *efficiency* reasons.
  - The declarations are analyzed by the compiler.
  - Dynamic semantic checks can be used to ensure properly constrained values are always assigned to subrange variables.
  - Might also use fewer bits to represent the values. (A `testScore` can be held in one byte.)

Type Classification

- The *Discrete*, *Rational*, *Real*, and *Complex* types together constitute the *Scalar* (also known as *Simple*) types.
  - An instance of each of these is a *single* value.
  - Yes, even *Rational* with its *Numerator* and *Denominator*.
  - Yes, even *Complex* with its *Real* and *Imaginary* parts.

- Beyond the *Scalar* types we also have the *Composite* types.
  - Also known as the *Nonscalar* types.

Composite Types

- Composite types include the following,
  - *Record*, *Union* — Collection of fields, each with own type.
  - *Array* — A mapping of an index type to a component type member.
  - *Set* — The mathematical notion of a set.
  - *Pointer* — A value that is a reference to an object.
  - *List* — A sequence of elements with no mapping or indexing.
  - *File* — Data (usually) on mass storage device.

Type Checking

- *Type Equivalence*
  - When are two types considered to be the same?
- *Type Compatibility*
  - When can an object of a given type be used in a given context?
- *Type Inference*
  - What is the type of an expression, given the types of the constituents and (sometimes) the context?

Type Checking

- *Type Equivalence* and *Type Compatibility* are ***not*** the same though they are closely related topics.
  - *Equivalence* means the two types are the ***same***.
  - *Compatibility* means one can use a value of one of the types when the other type is "expected".
- Generally, *Compatibility* is more useful as it tells one what one can ***do***.
- The terms are sometimes confused, so be careful.

Type Equivalence

- Type Equivalence is the determination if two types are the same or not.
- Doing this reasonably is *Non-obvious!*
- Two general ways to do this,
  - *Structural Equivalence* — Do the types "look" the same?
  - *Name Equivalence* — Do the types have the "same" name?

Structural Equivalence

- Roughly, two types are *Structurally Equivalent* if they have the same components put together in the same way.
- Algol-68, Modula-3, ML, and C (for *scalar* types) have structural equivalence.
- The exact definition varies from language to language.
  - Essentially, which differences are "important" and which aren't?

Structural Equivalence

- To determine structural equivalence, the compiler can compare two types "field by field".
  - Nested types are expanded into their definitions.
  - Expansion continues until the type is simply a list of field names of built-in types.
  - (Have to be careful about expanding *recursive* and *pointer-based* types, but this is not hard to deal with.)

Structural Equivalence Examples

- Are `A` and `B` to be considered the same type?
  - Most languages say *Yes*!
    - Same number of fields with same names and same (built-in) types.
- Are `B` and `C` the same type?
  - Most languages say *No*!
    - The fields have different names
  - ML says *Yes*!
    - Field names don't matter.

```
struct A {
 int a, b;
}

struct B {
 int a;
 int b;
}

struct C {
 int b;
 int a;
}
```

Structural Equivalence Examples

- Are `strA` and `strB` to be considered the same type?
  - Most languages say *No*!
    - Index ranges are clearly *different*.
- *But*, Ada and Fortran consider them *compatible*.
  - The index ranges are the same *size*.
  - Remember though that *compatible* is *not* the same as *equivalent*.

```
type strA =
  array [ 1 .. 10 ] of char;

type strB =
  array [ 0 ..  9 ] of char;
```

Structural Equivalence Examples

- Structural Equivalence can't distinguish *accidental* equivalence.
- Hard to believe that the user wanted the record types `student` and `school` to be equivalent.
- *Name Equivalence* would say they are different.

```
type student = rec
 name, address : string
 age : integer

type school = rec
 name, address : string
 age : integer

x : student
y : school

x := y;  // Error?
```

Name Equivalence

- Roughly, two types are *Name Equivalent* if they have the same name.
  - How can that be? Well, it's the *same name*; each definition introduces a new (unique) type.
  - Based on the idea that if one creates a new definition, it ought to be a new type.
- Java, C#, (standard) Pascal and Pascal descendents, Ada have name equivalence.
- C uses name equivalence for `structs` and `unions`.

Name Equivalence

- Name Equivalence seems pretty straightforward.
- *But* there are some subtleties.

- What about type *aliases*?
  - As in, e.g., C: `typedef oldType newType;`
  - Are `newType` and `oldType` the *same* type?
  - Or two *different* types that happen to have the same structure?
  - C says they are the *same* type.

Name Equivalence

- This *aliasing* can be useful when, e.g., building permission bits.
- `mode_t` is an alias for the unsigned 16-bit integer type.
- We can use `uint16_t` operations on `bits` even though it is a `mode_t`.

```
typedef uint16_t mode_t;

mode_t bits = S_IRUSR | S_IWUSR;

...

if ( bits & S_IRGRP ) ...
```

Name Equivalence

- On the other hand, there are times when type aliases probably should *not* be considered the same.
- Just because both `celsiusTemp` and `fahrenheitTemp` are aliases for `real` doesn't mean they are freely assignable.

```
type celsiusTemp = real;
type fahrenheitTemp = real;

var c : celsiusTemp;
var f : fahrenheitTemp;

...

f := c;  (* Oops! *)
```

Name Equivalence

- A language that says *aliased* types are *distinct* has *strict* name equivalence.
- If they are considered the *same*, the language has *loose* name equivalence.
- Most Pascal-derived languages have *loose* name equivalence.

Name Equivalence

- Ada takes a more liberal approach, having the concept of both *derived* types and *subtypes*.
- A *subtype is* compatible with its base type.
  - `mode_t` is an unsigned 16-bit integer.
- A *derived* type *is not* compatible with its base type.
  - `celsiusTemp` and `fahrenheitTemp` are *different* types.

```
subtype mode_t is
  integer range 0..2**16-1;

type celsiusTemp is
  new integer;

type fahrenheitTemp is
  new integer;
```

Type Equivalence Summary Example

- Under *Structural Equivalence*, all of the variables have the same type.
- Under *Strict Name Equivalence*,
  - `p` and `q` have the same (anonymous) type.
  - `r` and `u` have the same type (`alink`).
- Under *Loose Name Equivalence*,
  - `p` and `q` have the same (anonymous) type.
  - `r`, `s`, and `u` have the same type (`alink` and `blink` are considered the same type).

```
type cell = ...
type alink = pointer to cell
type blink = alink

p, q : pointer to cell
r    : alink
s    : blink
t    : pointer to cell
u    : alink
```

Type Conversions and Casts

- When using a value of one type in a context that requires another type, a *type conversion* (or *cast*) is required.
- Three cases …

① The types are *structurally equivalent*, but the language uses *name equivalence*. The underlying bits are the same, so the "conversion" is merely notation in the code.

```
n : integer;     -- 32-bit signed value
c : celsiusTemp; -- integer

...

n := integer( c );     -- no conversion or check required
c := celsiusTemp( n ); -- no conversion or check required
```

Type Conversions and Casts

② The types represent different values, but intersecting values are represented the same. Again, the underlying bits are the same but a *dynamic type check* may be required to ensure the value is acceptable.

```
n : integer;      -- 32-bit signed value
t : test_score:   -- 0 .. 100

...

t := test_score( n );  -- no conversion but dynamic check required
n := integer( t );     -- no conversion or check required
```

Type Conversions and Casts

③ The types have different representations but a correspondence can be determined. Since the bits are different, a *dynamic conversion* must be done and the resulting value may have to be *dynamically type checked*.

```
n : integer;      -- 32-bit signed value
r : real;         -- IEEE-754 double precision

...

r := real( n );         -- dynamic conversion but no check required
n := integer( r );      -- dynamic conversion and check required
```

Type Conversion Examples Summary

```
n : integer;     -- 32-bit signed value
r : real;        -- IEEE-754 double precision
t : test_score:  -- 0 .. 100
c : celsiusTemp; -- integer

...

t := test_score( n );  -- no conversion but dynamic check required
n := integer( t );     -- no conversion or check required

r := real( n );        -- dynamic conversion but no check required
n := integer( r );     -- dynamic conversion and check required

n := integer( c );     -- no conversion or check required
c := celsiusTemp( n ); -- no conversion or check required
```

Non-Converting Type Casts

- More specifically, interpreting the *bits* of a value of one type as if they were the *bits* of a value of another type.
  - *Type Punning*!
- Commonly occurs in, e.g., memory allocation routines.
  - Memory itself is *untyped* but when a heap allocation occurs, the newly-allocated block is interpreted as a value of a type.
- Also used to extract bits from a packed format.
  - E.g., interpret a FP value as an integer to make sign, exponent, and significand "fields" accessible as bits.

Non-Converting Type Casts

- Though it's called a *cast*, it does *not ever* result in underlying bits being altered in any way.
- Ada provides the built-in generic subroutine called `unchecked_conversion` to do this.
- C permits *any* non-converting type cast. (YAFIYGI!), but C++ tries to be sane by providing alternative mechanisms:
  - `reinterpret_cast` — the non-converting type cast.
  - `static_cast` — type-converting cast.
  - `dynamic_cast` — a cast that has to be checked dynamically.
  - `const_cast` — (to remove `const`-ness).

Non-Converting Type Casts

- *Any* use of a non-converting type cast is a *potentially dangerous* subversion of the language's type system.
- In a *weakly-typed* language, these can be difficult to find and resolve when they go wrong.
- In a *strongly-typed* language, at least using the explicit notation makes it clear that something dangerous is happening.

Type Compatibility

- Most languages do *not* require type *equivalency* in every context.
- However, at least type *compatibility* is required.
- For example,
  - The type of the RHS of an assignment (the expression) must be *compatible* with the type of the LHS (the destination).
  - In an expression, the types of *operands* must be *compatible* with some common type that supports the *operator*.
  - The type of an *argument* must be *compatible* with the type of the corresponding *formal parameter*.

Type Compatibility

- The definition of type compatibility varies greatly from language to language.
  - Ada is quite restrictive
    - Types are compatible if-and-only-if ① they are *equivalent*, ② one is a subtype of the other, or ③ they are arrays of the same number and types of elements.
  - Pascal is somewhat more lenient.
    - Beyond the previous restrictions, ④ an `integer` can be used where a `real` is expected.

Coercion

- If a value of a *non-equivalent* type can be used in a given context, it must be *automatically* and *implicitly* converted to the required type.
- This conversion is known as *(Type) Coercion*.
  - May require dynamic semantic checks or bit-level conversions to ensure the proper and correct result.
- C performs many kinds of coercion.
  - E.g., mixing of numeric types in expressions.
  - But does *no* dynamic semantic checks!

Coercion Examples in C

```
short int s;            // Usually 16 bits
unsigned long int usl;  // Usually 64 bits
char c;                 // May be signed or unsigned
float f;                // Usually IEEE-754 single-precision
double d;               // Usually IEEE-754 double-precision

s = usl; // usl's low-order bits are interpreted as a signed number.

usl = s; // s is sign-extended to the length of usl, then interpreted as an unsigned number.

// c is either sign- or zero-extended to the length of s and then interpreted as a signed number.
s = c;

// usl is converted to floating point.  f has fewer significant bits, so precision may be lost.
f = usl;

d = f;   // f is converted to the longer format.  No precision is lost

// d is converted to the shorted format;  precision may be lost.
// If d won't "fit" in f, the result is undefined, but NOT a dynamic semantic error.  Ha!
f = d;
```

Coercion

- Coercion is a *weakening* of the type system — and therefore somewhat controversial.
- It happens *implicitly* and there is *no* indication of *user intent*.
- On the other hand, coercion is *incredibly useful*.
  - It seems a natural way to support *abstraction* and *extensibility*.
- Most scripting languages support a wide variety of coercion.
- Statically-typed languages differ greatly in what gets coerced.

Coercion

- Ada will coerce nothing but explicit constants, subranges, and (in some cases) arrays with same type of elements.
- Pascal will coerce integers to floating-point values in expressions and assignments.
  - But not the other way!
- Fortran will coerce floating-point values to integers, but only in an assignment (with a potential loss of precision).
- C will do all of the previous as well as coercions on arguments to function calls.

Coercion

- C++ provides the most extreme example of coercion in a statically-typed language.
- C++ allows the *user* to define coercion operations when defining a *class*.
- The rules for applying these coercions interact in complicated ways with the rules for resolving overloading.
- *Very* flexible, but also *very tricky* to understand and use correctly.

Overloading and Coercion

- *Overloading* and *Coercion* can sometimes be used to similar effect.
- For example, in `a + b`, `+` might be overloaded to mean either integer addition or floating-point addition.
- In a language with no coercion, `a` and `b` would have to be both integer or both floating point.
- With coercion, `+` is floating-point addition whenever *at least one* of `a` and `b` is floating point, otherwise it's integer addition.

[*Conversion, Non-Converting Casting, and Coercion …*]

- Be careful! These are three *separate* kinds of operations even though they all result in a change of type.
- Conversion and Non-Converting Casting are *explicitly* written by the user.
  - Conversion *may* or *may not* change the underlying bits. Non-Converting Casting does *not* change the bits.
- Coercion is *implicitly* applied by the compiler.
  - The underlying bits *may* or *may not* be changed.

Universal Reference Types

- *Universal Reference Types (URT)* are useful when writing general-purpose *container* objects holding references to other objects.
  - Container objects include, e.g., lists, stacks, queues, sets.
- For C and C++, it's `void *`. In Java, it's `Object`. In C#, it's `object`, …
- Arbitrary L-values can be assigned into an object of the URT with no concern for type safety.
- Safely getting the value out again is trickier.

Universal Reference Types

- For example, a URT holding a floating-point value should not be assigned to an integer variable.
  - The underlying representations are incompatible.
- One solution is to make objects *self-descriptive*.
  - This is common in object-oriented languages as it's needed for *dynamic method binding*.
- In Java and C# a type cast is required and a dynamic error will result if a type clash is detected.

Type Inference

- Type *Checking* (with *Conversion* and *Coercion*) ensures that the *operands* in an expression and the *arguments* in a function application have the correct types.
- Type *Inference* is used to determine the type of the *result* of the expression evaluation.
- This *inference* is often easy …
  - The type of a comparison is normally Boolean.
  - The type of an operation is often the type of the operands.
  - The type of a function application is given by the function's declaration.
  - The type of an assignment is the type of the L-value.

Type Inference

- Sometimes, however, the inference is not so easy.
- Operations on subranges and composite objects often do not preserve the type of the operand.
- For example, what is the type of `a + b` given the declarations in ①?

```
type                                    ①
 Atype =  0 .. 20;
 Btype = 10 .. 20;

var
 a : Atype;
 b : Btype;

// type( a + b ) → ?
```

Type Inference

- Possible values ranges from `10` to `40` so it can't be either `Atype` or `Btype`.
- The usual answer is the subrange's base type, here `integer`.
- If the value of `a + b` were assigned to a variable of `Atype` or `Btype`, a *dynamic semantic check* may have to be generated to ensure that an out-of-bounds value does not get assigned.

```
type                                    ①
 Atype =  0 .. 20;
 Btype = 10 .. 20;

var
 a : Atype;
 b : Btype;

// type( a + b ) → ?
```

Type Inference

- A similar situation occurs in languages with a *set* datatype.
- Set *union*, *intersection*, and *difference* operations on sets of discrete values can cause results that are out-of-type for any of the operand types.
- The compiler can infer an anonymous set of the base types of the elements of the operands.
  - These base types must be *compatible*.
  - As with the subrange case, *dynamic semantic checks* may have to be generated to ensure any assignments are compatible.

Parametric Polymorphism

- For many compiled languages, explicit parametric polymorphism (*generics*) allows type parameters when declaring a routine or a class.
  - Ada, C++ (called *templates*), Eiffel, Java, C#, Scala, …
- It's irritating that both ① and ② have to be written when they differ only in the type (`integer` vs. `long_float`).

①
```
function min( x, y: integer )
 return integer is
begin
 if x < y then return x;
 else return y;
 end if;
end min;
```

②
```
function min( x, y: long_float )
 return long_float is
begin
 if x < y then return x;
 else return y;
 end if;
end min;
```

Parametric Polymorphism

- The `generic` version at left abstracts the type into a separate parameter.
- The definition can be used to create any number of type-specific `min` functions.
- (An appropriate ordering function `"<"` is required.)

Ada

```
generic
 type T is private;
 with function "<"( x, y: T ) return Boolean;
function min( x, y: T ) return T;

function min( x, y: T ) return T is
begin
 if x < y then return x;
 else return y;
 end if;
end min;

function intMin is new min( integer, "<" );
function realMin is new min( long_float, "<" );
function stringMin is new min( string, "<" );
function dataMin is new min( data, dataPrecedes );
```

Parametric Polymorphism

- In C++, the `template` mechanism is used for this.
- The definition can be used to create any number of type-specific `min` functions.

C++

```
#include <iostream>

//--------------------------------------
template <typename T>
inline T min( T a, T b )
{
  return a < b ? a : b;
}

//--------------------------------------
using namespace std;

int main()
{
  // Creates minimum<int>
  cout << min( 4, 5 ) << endl;

  // Creates minimum<double>
  cout << min( 4.3, 5.4 ) << endl;

  // Have to explicitly specify how to
  // instantiate the template when it's
  // ambiguous.
  cout << min<double>( 4, 5.4 ) << endl;
}
```

Parametric Polymorphism

- In C++, the `template` mechanism is used for this.
- The definition can be used to create any number of type-specific `min` functions.
- Ambiguous case generates error if not clarified with explicit type.

```
In function 'int main()':
error: no matching function for call to 'min(int, double)'
   cout << min( 4, 5.4 ) << endl;
                        ^
note: candidate: template<class T> T min(T, T)
 inline T min( T a, T b )
          ^~~~~~
note:   template argument deduction/substitution failed:
note:   deduced conflicting types for parameter 'T' ('int' and
'double')
   cout << min( 4, 5.4 ) << endl;
```

C++

```
#include <iostream>

//--------------------------------------
template <typename T>
inline T min( T a, T b )
{
  return a < b ? a : b;
}

//--------------------------------------
using namespace std;

int main()
{
  // Creates minimum<int>
  cout << min( 4, 5 ) << endl;

  // Creates minimum<double>
  cout << min( 4.3, 5.4 ) << endl;

  // Have to explicitly specify how to
  // instantiate the template when it's
  // ambiguous.
  cout << min<double>( 4, 5.4 ) << endl;
}
```

Generics in Ada, C++, Java, C#, Lisp, ML

	Explicit (generics)				Implicit	
	Ada	C++	Java	C#	Lisp	ML
Applicable to	subroutines, modules	subroutines, classes	subroutines, classes	subroutines, classes	functions	functions
Abstract over	types; subroutines; values of arbitrary types	types; enum, int, and pointer constants	types only	types only	types only	types only
Constraints	explicit (varied)	implicit	explicit (inheritance)	explicit (inheritance)	implicit	implicit
Checked at	compile time (definition)	compile time (instantiation)	compile time (definition)	compile time (definition)	run time	compile time (inferred)
Natural implementation	multiple copies	multiple copies	single copy (erasure)	multiple copies (reification)	single copy	single copy
Subroutine instantiation	explicit	implicit	implicit	implicit	—	—

Parametric Polymorphism

- Java and C# provide generics purely for polymorphism.
  - Java's generics were designed for backwards compatibility with earlier versions of Java as well as existing virtual machines and libraries.
  - C#'s were anticipated from the start.
- C++ templates are intended for almost *any* programming that requires *substantially similar* but *not identical* copies of an abstraction.

Equality Testing

- For *primitive* types such as integers, reals, booleans, and characters, *equality testing* is simple.
  - Bit-by-bit comparisons and copying get the job done.
- Beyond that, the operation gets trickier.
  - For example, consider comparing two character *strings*.
  - Does `str1 == str2` mean `str1` and `str2` …
    - Are *aliases* for each other? Are *bit-wise* identical across their entire lengths? Contain the *same sequence* of characters? Would *appear the same* if printed?

Equality Testing

- *Aliases* for each other is kind of strict, but at least one knows that equal is truly equal.
- *Bit-wise* identical at first seems reasonable, but what about, e.g., garbage in the storage area after the "length" of the string?
- *Same sequence* of characters could be tricky to check given that in some encodings there are multiple ways to represent the same character.
- *Appear the same* can depend on external representations that are not necessarily fixed. Equality might change!

Equality Testing

- In the presence of *references*, should expressions be considered *equal* only if they *refer* to the *same object*?
  - This is known as *shallow* comparison.
  - A generally simple test.
- Or, is there another sense of *equal* such that two *different* objects can be considered *equal*?
  - This is known as *deep* comparison.
  - More complex test and in the presence of complex data structures could require *recursive traversal*.

[*Equality Testing in Scheme …*]

- Scheme (a Lisp derivative) offers multiple predicates to test equality.
- While there are some implementation dependencies, portable behavior can be guaranteed.
- `eq?` is strictest.
- `eqv?` is value oriented.
- `equal?` checks the structure recursively.

```
(eq? a b)    ; Are a and b the same object?
(eqv? a b)   ; Are a and b semantically equivalent?
(equal? a b) ; Do a and b have the same structure?

(eq? #t #t)          ; True!  The same objects.
(eq? 'foo 'foo)      ; True!  The same objects.
(eq? '(a b) '(a b))  ; False!  Different cons cells.

(eq? 2 2)            ; Implementation dependent!
(eqv? 2 2)           ; True!  Same (numeric) value.

(eq?  "foo" "foo")   ; Implementation dependent!
(eqv? "foo" "foo")   ; Implementation dependent!
(equal? "foo" "foo") ; True!  Same characters in strings.
```

Type Systems Summary …

- Types serve *two* principal purposes …
  - They provide a context for many purposes, freeing the programmer from having to do so explicitly.
  - They allow the compiler to catch a wide variety of common programming errors.
- Types may be considered on the basis of …
  - (Denotational) … their values.
  - (Structural) … their substructure.
  - (Abstractional) … the operations they support.

Type Systems Summary …

- A Type System consists of …
  - A set of *builtin types*.
  - A mechanism to define *new types*.
  - Rules for …
    - *Type equivalence* (when two values or named objects have the same type).
    - *Type compatibility* (when a value of one type may be used in a context that expects another type).
    - *Type inference* (the determination of the type of an expression from its components or (sometimes) the context).

Type System Summary …

- A language is *strongly typed* if it never allows an operation to be applied to an object that does not support it.
- A language is *statically typed* if it enforces strong typing at compile time.
- *Polymorphism* allows a routine to operate on values of multiple types as long as only operations supported by those types are used.
- *Parametric Polymorphism* provides the types of the operands as additional parameters.

Review Questions (1)

1. What purpose(s) do types serve in a programming language?
2. What does it mean for a language to be *strongly typed*? *Statically typed*? What prevents, say, C from being strongly typed?
3. Name two programming languages that are strongly but dynamically typed?
4. What is a *type clash*?
5. Discuss the differences among the *denotational*, *structural*, and *abstraction-based* views of types.
6. What does it mean for a set of language features (e.g., a type system) to be *orthogonal*?

Review Questions (2)

7. What are *aggregates*?
8. What are *option* types? What purpose do they serve?
9. What is *polymorphism*? What distinguishes its *parametric* and *subtype* varieties? What are *generics*?
10. What is the difference between *discrete* and *scalar* types?
11. Give two examples of languages that lack a Boolean type. What do they use instead?

Review Questions (3)

12. In what ways may an enumeration type be preferable to a collection of named constants? In what ways may a subrange type be preferable to its base type? In what ways may a string be preferable to an array of characters?
13. What is the difference between *type equivalence* and *type compatibility*?
14. Discuss the comparative advantages of *structural* and *name* equivalence for types. Name three languages that use each approach.

Review Questions (4)

15. Explain the difference between *strict* and *loose* name equivalence.
16. Explain the distinction between *derived* types and *subtypes* in Ada.
17. Explain the differences among *type conversion*, *type coercion*, and *nonconverting type casts*.
18. Summarize the arguments for and against coercion.
19. Under what circumstances does a type conversion require a run-time check?
20. What purpose is served by *universal reference types*?
21. What is *type inference*? Describe three contexts in which it occurs.

Review Questions (5)

22. Under what circumstances does an ML compiler announce a type clash?
23. Explain how the type inference of ML leads naturally to polymorphism.
24. Why do ML programmers often declare the types of variables, even when they don't have to?
25. What is *unification*? What is its role in ML?
26. Explain the distinction between implicit and explicit parametric polymorphism. What are their comparative advantages?

Review Questions (6)

27. What is *duck typing*? What is its connection to polymorphism? In what languages does it appear?
28. Explain the distinction between overloading and generics. Why is the former sometimes called *ad hoc* polymorphism?
29. What is the principal purpose of generics? In what sense do generics serve a broader purpose in C++ and Ada than they do in Java and C#?
30. Under what circumstances can a language implementation share code among separate instances of a generic?

Review Questions (7)

31. What are *container* classes? What do they have to do with generics?
32. What does it mean for a generic parameter to be *constrained*? Explain the difference between explicit and implicit constraints. Describe how interface classes can be used to specify constraints in Java and C#.
33. Why will C# accept `int` as a generic argument, but C# won't?
34. Under what circumstances will C++ instantiate a generic function implicitly?
35. Why is equality testing more subtle than it first appears?