

Type Checking

- *Type checking* deals with a number of topics, generally dealing with determining an object's type as well as the context in which that object is (or can be) used
- General outline:
 - Type equivalence
 - Type compatibility
 - Conversion/casting
 - Nonconverting casts
 - Coercion
 - Type inference
- Object-oriented programmers beware — while classes are indeed the OO version of a *type*, many of the following issues may feel foreign to you because of OO's specialized semantics

Type Equivalence

- *Structural equivalence*: equivalent if built in the same way (same parts, same order)
- *Name equivalence*: distinctly named types are always different
- Structural equivalence questions
 - What parts constitute a structural difference?
 - Storage: record fields, array size
 - Naming of storage: field names, array indices
 - Field order
 - How to distinguish between intentional vs. incidental structural similarities?
 - An argument for name equivalence: “They're different because the programmer said so; if they're the same, then the programmer won't define two types for them.”

Type Equivalence Issues & Non-Issues

- Would record types with identical fields, but different name order, be structurally equivalent?

```
type PascalRec = record a : integer; b : integer end;

val MLRec = { a = 1, b = 2 };
val OtherRec = { b = 2, a = 1 };
```

- When are arrays with the same number of elements structurally equivalent?

```
type str = array [1..10] of integer;
type str = array [1..2 * 5] of integer;
type str = array [0..9] of integer;
```

- Moot point for languages that don't allow variations in array indices!

Alias Types and Name Equivalence

- *Alias types* are types that purely consist of a different name for another type

```
TYPE Stack_Element = INTEGER;
TYPE Level = INTEGER;
TYPE Celsius = REAL;
TYPE Fahrenheit = REAL;
```

- Should INTEGERS be assignable to a Stack_Element? How about Levels?
- On the flip side, can a Celsius and Fahrenheit be assigned to each other?

- *Strict name equivalence*: aliased types are distinct
- *Loose name equivalence*: aliased types are equivalence
- Ada allows additional explicit equivalence control:

```
subtype Stack_Element is integer;
type Celsius is new real;
type Fahrenheit is new real;
```

- Modula-3's BRANDED keyword explicitly marks a type as distinct at all times, regardless of structural equivalence

Type Conversion

- Certain contexts in certain languages may require exact matches with respect to types:
 - $aVar := anExpression$
 - $value1 + value2$
 - $foo(arg1, arg2, arg3, \dots, argN)$
- *Type conversion* seeks to follow these exact match rules while allowing programmers some flexibility in the values used
 - Using structurally-equivalent types in a name-equivalent language
 - Types whose value ranges may be distinct but intersect (e.g. subranges)
 - Distinct types with sensible/meaningful corresponding values (e.g. integers and floats)
- Explicit conversions are typically called *type casts*
- Type conversions may sometimes add code to a program:
 - Code to actually perform the conversion
 - Code to perform semantic checks on the conversion result

Type Casting Syntax

- Ada:

```
n : integer;  
r : real;  
...  
r := real(n);
```
- C/C++/Java:

```
// Sample is specific to Java, but shares common syntax.  
Object n;  
String s;  
...  
s = (String)n;
```
- Some SQL flavors:

```
-- Timestamp is a built-in data type; charField is  
-- a varchar (string) field of some table.  
select charField::timestamp from...
```

Nonconverting Type Casts

- Type casts that explicitly preserve the internal bit-level representation of values
- Common in manipulating allocated blocks of memory
 - Same block of memory may be viewed as arrays of characters, integers, or even records/structures
 - Block of memory may be read from a file or other external source that is initially viewed as a “raw” set of bytes
- Ada: explicit *unchecked_conversion* subroutine

```
function cast_float_to_int is
  new unchecked_conversion(float, integer);
```
- C/C++ (but not Java!): pointer games

```
void *block; // Gets loaded up with some data, say from a file.
Record *header = (Record *)block; // Record is some struct type.
```
- C++: explicit cast types *static_cast*, *reinterpret_cast*, *dynamic_cast*

```
int i = static_cast<int>(d); // Assume d is declared as double.
Record *header = reinterpret_cast<Record *>(block);
Derived *dObj = dynamic_cast<Derived *>(baseObj); // Derived is a subclass of Base.
```

Type Coercion

- Sometimes absolute type equivalence is too strict; *type compatibility* is sufficient
- Type equivalence vs. type compatibility in Ada (strict):
 1. Types must be equivalent
 2. One type must be a subtype of another, or both are subtypes of the same base type
 3. Types are arrays with the same sizes and element types in each dimension
- Pascal extends slightly, also allowing:
 - Base and subrange types are cross-compatible
 - Integers may be used where a real is expected
- *Type coercion* is an implicit type conversion between compatible but not necessarily equivalent types

Type Coercion Issues

- Sometimes viewed as a weakening of type security
 - Allows mixing of types without explicit indication of intent
 - Opposite end of the spectrum: C and Fortran
 - Allow interchangeable use of numeric types
 - Fortran: arithmetic can be performed on entire arrays
 - C: arrays and pointers are roughly interchangeable

- C++ adds programmer-extensible coercion rules

```
class ctr {
public:
    ctr(int i = 0, char* x = "ctr") { n = i; strcpy(s, x); }
    ctr& operator++(int) { n++; return *this; }
    operator int() { return n; } // Coercion to int
    operator char*() { return s; } // Coercion to char *
private:
    int n; char s[64];
};
```

More Type Coercion Thoughts

- Overloading and type coercion: may feel similar but with real semantic differences
 - Case in point — overloaded “+” vs. coercing “+”
- How to handle constants — is “5” an int or a float? What is the base type of a *nil* or *NULL* reference?
 - Constants may be viewed as having more than one possible type (and therefore are overloaded) and coerced as needed
 - Ada makes this explicit and formal: constants have distinct types from variables (*universal_integer* vs. *integer*; *universal_real* vs. *real*)
 - Allows use of constants in any derived type
- “Generic” objects: void * (C/C++), any (Clu), address (Modula-2), Object (Java)
 - Nice for abstraction (e.g. data structures, translation from memory or I/O)
 - May require self-descriptive entities (*type tags*) — values “know” about their own type information, allowing runtime checking of type casts

Type Inference

- *Type inference* refers to the process of determining the type of an arbitrarily complex expression
- Generally not a huge issue — most of the time, the type for the result of a given operation or function is clearly known, and you just “build up” to the final type as you evaluate the expression
- In languages where an assignment is also an expression, the convention is to have the “result” type be the type of the left-hand side
- But, there are occasional issues, specifically with subrange and composite types

Type Inference Special Case 1

- *Subranges* — in languages that can define types as subranges of base types (Ada, Pascal), type inference can be an issue:

```
type Atype = 0..20; Btype = 10..20;
var a : Atype; b : Btype; c : ????
```

 - `c := a + b;`
 - What should `c`'s type be? Easy answer: always go back to the base type (integer in this case)
- What if the result of an expression is assigned to a subrange?

```
a := 5 + b; (* Where a and b are defined as above *)
```

 - The primary question is bounds checking — operations on subranges can certainly produce results that break away from their defined bounds
 - Static checks: include code that infers the lowest and highest possible results from an expression
 - Dynamic check: static checks are not always possible, so the last resort is to check the result at runtime

Type Inference Special Case 2

- *Composite types* — What is the type of operators on arrays? We know it's an array, but what specifically? (particularly for languages where the index range is part of the array definition)
 - Case in point: strings in languages where strings are exactly character arrays (Pascal, Ada)
- Another tricky composite type: sets. In languages that encode a base type with a set (e.g. set of integer), what is the “type” of unions, intersections, and differences of sets?
 - Particularly tricky when a set is combined with a subrange

```
var A : set of 1..10; B : set of 10..20; C : set of 1..15; i : 1..30;  
... C := A + B * [1..5, i];
```
 - Same as subrange handling: static checks are possible in some cases, but dynamic checks are not completely avoidable

Types in ML: Type Inference Extreme

- Full-blown type inference
- The “feel” of untyped declarations without losing the checks provided by strong typing
- Accommodates polymorphism

```
fun fib n =  
  let fun fib_helper f1 f2 i =  
        if i = n then f2 else fib_helper f2 (f1 + f2) (i + 1)  
      in  
        fib_helper 0 1 0  
      end;
```

- ML figures out that *fib* is a function that takes an integer and retains an integer through a series of deductions, usually starting with any literals in the code

ML Type Correctness = Type Consistency

- The key to ML's type inference is the absence of inconsistency or ambiguity. Pitfalls include:
 - Arithmetic operations that switch between real and integer operands
 - Functions that can go “either way” — this will require explicit type declarations:

```
fun square x = x * x; (* Defaults to int -> int *)
```
- *But* this does not rule out polymorphism. If an operation is polymorphic, then the function is also polymorphic:
 - Easy example: equality has type `'a * 'a -> bool`
 - Not so obvious but works just fine thank you:

```
fun twice f x = f (f x);  
twice (fn x => x / 2.0) 1.0;  
twice (fn x => x ^ "ee") "whoop";
```

Type Unification

- Part of ML's type inference is *unification* — composing or combining multiple types in a consistent manner
 - Say E1 has type `'a * int` and E2 has type `string * 'b`
 - `if x then E1 else E2` can be inferred as having type `string * int`
- The type system is completely orthogonal with lists

```
fun append l1 l2 =  
  if l1 = nil then l2 else hd (l1) :: append t1(l1) l2;
```

```
fun member x l =  
  if l = nil then false else  
    if x = hd (l) then true else member x t1(l);
```

Other ML Type Notes

- Tuple types (a, b, c) allow functions to be fixed at having a single argument
 - “Multiple arguments” can be expressed either as a tuple, or
 - ...in the cooler ML way (which we have been using), by *currying*: functions with arguments are themselves functions, and can be given additional arguments (this is the argument-without-parentheses notation that you have been seeing)
- ML has records $\{ \text{name} \Rightarrow \text{value}, \dots, \text{name}_n \Rightarrow \text{value}_n \}$ which operate based on structural equivalence independent of field name order
- New types can be synthesized using `datatype` and a special notation for constructors