G22.2110-003 Programming Languages - Fall 2012

Lecture 6

Thomas Wies

New York University

Review

Last week

- Functional Languages
- Lambda Calculus
- Scheme

Outline

- Types

Sources:
PLP, ch. 7

Types

What is a type?

- A type consists of a set of values
- The compiler/interpreter defines a mapping of these values onto the underlying hardware.

Types

What purpose do types serve in programming languages?

Types

What purpose do types serve in programming languages?

- Implicit context for operations
  - Makes programs easier to read and write
  - Example: `a + b` means different things depending on the types of `a` and `b`.
- Constrain the set of correct programs
  - Type-checking catches many errors

Type Systems

A *type system* consists of:

- a mechanism for defining types and associating them with language constructs
- a set of rules for:
  - *type equivalence*: when do two objects have the same type?
  - *type compatibility*: where can objects of a given type be used?
  - *type inference*: how do you determine the type of an expression from the types of its parts

Type Systems

A *type system* consists of:

- a mechanism for defining types and associating them with language constructs
- a set of rules for:
  - *type equivalence*: when do two objects have the same type?
  - *type compatibility*: where can objects of a given type be used?
  - *type inference*: how do you determine the type of an expression from the types of its parts

What constructs are types associated with?

Type Systems

A *type system* consists of:

- a mechanism for defining types and associating them with language constructs
- a set of rules for:
  - *type equivalence*: when do two objects have the same type?
  - *type compatibility*: where can objects of a given type be used?
  - *type inference*: how do you determine the type of an expression from the types of its parts

What constructs are types associated with?

- Constant values
- Names that can be bound to values
- Subroutines (sometimes)
- More complicated expressions built up from the above

Type Checking

Type checking is the process of ensuring that a program obeys the type system's type compatibility rules.

A violation of the rules is called a *type clash*.

Languages differ in the way they implement type checking:

- strong vs weak
- static vs dynamic

Strong vs Weak Typing

- A strongly typed language does not allow variables to be used in a way inconsistent with their types (no loopholes)
- A weakly typed language allows many ways to bypass the type system (e.g., pointer arithmetic)

C is a poster child for the latter. Its motto is: "Trust the programmer".

Static vs Dynamic Type Systems

Static vs Dynamic

- Static
  - Variables have types
  - Compiler ensures that type rules are obeyed at compile time
  - Ada, Pascal, ML, Scala
- Dynamic
  - Variables do not have types, values do
  - Compiler ensures that type rules are obeyed at run time
  - Lisp, Scheme, Smalltalk, scripting languages

A language may have a mixture: Java has a mostly static type system with some runtime checks.

Pros and cons

- static: faster (dynamic typing requires run-time checks), easier to understand and maintain code, better error checking
- dynamic: more flexible, easier to write code

Polymorphism

Polymorphism allows a single piece of code to work with objects of multiple types.

- *Parametric polymorphism*: types can be thought of as additional parameters
  - *implicit*: often used with dynamic typing: code is typeless, types checked at run-time (Lisp, Scheme) - can also be used with static typing (ML)
  - *explicit*: templates in C++, generics in Java and Scala
- *Subtype polymorphism*: the ability to treat a value of a subtype as a value of a supertype
- *Class polymorphism*: the ability to treat a class as one of its superclasses (special case of subtype polymorphism)

Parametric polymorphism example

Scheme

```
(define (length xs)
  (cond
    ((null? l) 0)
    (#t (+ (length (cdr xs)) 1))))
```

The types are checked at run-time.

ML

```
fun length xs =
  if null xs
  then 0
  else 1 + length (tl xs)
```

How can ML be statically typed and allow polymorphism?
It uses *type variables* for the unknown types.
The type of this function is written `'a list -> int`.

Assigning types

Programming languages support various methods for assigning types to program constructs:

- *determined by syntax*: the syntax of a variable determines its type (FORTRAN 77, ALGOL 60, BASIC)
- *no compile-time bindings*: dynamically typed languages
- *explicit type declarations*: most languages

Types: Points of View

Denotational

- type is a set T of values
- value has type T if it belongs to the set
- object has type T if it is guaranteed to be bound to a value in T

Constructive

- type is either *built-in* (int, real, bool, char, etc.) or
- *constructed* using a *type-constructor* (record, array, set, etc.)

Abstraction-based

- Type is an *interface* consisting of a set of operations

Scalar Types Overview

- *discrete types*
  must have clear successor, predecessor
  - *integer types*
    often several sizes (e.g., 16 bit, 32 bit, 64 bit, arbitrary precision)
    sometimes signed and unsigned variants (e.g., C/C++, ADA, C#)
  - *enumeration types*
- *floating-point types*
  typically 64 bit (double in C); sometimes 32 bit as well (float in C)
- *rational types*
  used to represent exact fractions (SCHEME, LISP)
- *complex*
  FORTRAN, SCHEME, LISP, C 99, C++ (in STL)

Other simple types

- boolean
  Common type; C had no boolean until C 99
- character, string
  - some languages have no character data type (e.g., JavaScript)
  - internationalization support
    - Java: UTF-16
    - C++: 8 or 16 bit characters; semantics implementation dependent
  - string mutability
    most languages allow it, Java does not.
- empty or trivial types are often used as return type of procedures;
  `void`: (C, Java) represents the absence of a type
  `unit`: (ML, Haskell,Scala) a type with one value: `()`
  `Nothing`: (Scala) the empty type

Enumeration types: abstraction at its best

- trivial and compact implementation:
  values are mapped to successive integers
- very common abstraction: list of names, properties
- expressive of real-world domain, hides machine representation

Examples (ADA):

```
type Suit is (Hearts, Diamonds, Spades, Clubs);
type Direction is (East, West, North, South);
```

Order of list means that `Spades > Hearts`, etc.

Contrast this with C#:

> *"arithmetics on enum numbers may produce results in the underlying representation type that do not correspond to any declared enum member; this is not an error"*

Enumeration types and strong typing

```
type Fruit is (Apple, Orange, Grape, Apricot);
type Vendor is (Apple, IBM, HP, Dell);

My_PC : Vendor;
Dessert : Fruit;
...
My_PC := Apple;
Dessert := Apple;
Dessert := My_PC;  -- error
```

`Apple` is *overloaded*. It can be of type `Fruit` or `Vendor`.
Overloading is allowed in C#, Java, Ada
Not allowed in Pascal, C

Subranges

ADA and PASCAL allow types to be defined which are subranges of existing discrete types.

```
type Sub is new Positive range 2..5;
subtype workday is weekday range mon..fri;
V: Sub -- Ada

type sub = 2..5;
type workday = mon..fri
var v: sub; (* Pascal *)
```

Assignments to these variables are checked at runtime:

```
V := I + J;  -- runtime error if not in range
```

Composite Types

- arrays
- records
- variant records, unions
- pointers, references
- function types
- lists
- sets
- maps

Composite Literals

Does the language support these?

- array aggregates

```
A := (1, 2, 3, 10);              -- positional
A := (1, others => 0);           -- for default
A := (1..3 => 1, 4 => -999);     -- named
```

- record aggregates

```
R := (name => "NYU", zipcode => 10012);
```

Type checking and inference

- *Type checking:*
  - Variables are declared with their type.
  - Compiler determines if variables are used in accordance with their type declarations.
- *Type inference:* (ML, Haskell, Scala)
  - Variables are declared, but not their type.
  - Compiler determines type of a variable from its usage/initialization.

In both cases, type inconsistencies are reported at compile time.

```
fun f x =
  if x = 5  (* There are two type errors here *)
  then hd x
  else tl x
```

Type equivalence

Name vs structural

- *name equivalence*

  two types are the same only if they have the same name
  (each type definition introduces a new type)

  - *strict*: aliases (i.e. declaring a type equal to another type) are distinct
  - *loose*: aliases are equivalent

- *structural equivalence*

  two types are equivalent if they have the same structure

Most languages have mixture, e.g., C: name equivalence for records (structs), structural equivalence for almost everything else.

Type equivalence example

Name equivalence in Ada:

```
type t1 is array (1 .. 10) of boolean;
type t2 is array (1 .. 10) of boolean;
v1: t1;  v2: t2;
-- v1, v2 have different types
```

Structural equivalence in ML:

```
type t1 = { a: int, b: real };
type t2 = { b: real, a: int };
(* t1 and t2 are equivalent types *)
```

Accidental structural equivalence

```
type student = {
  name: string,
  address: string
}

type school = {
  name: string,
  address: string
}

type age = float;
type weight = float;
```

With structural equivalence, we can accidentally assign a `school` to a `student`, or an `age` to a `weight`.

Type conversion

Sometimes, we want to convert between types:

- if types are structurally equivalent, conversion is trivial (even if language uses name equivalence)
- if types are different, but share a representation, conversion requires no run-time code
- if types are represented differently, conversion may require run-time code (from `int` to `float` in C)

A *nonconverting type cast* changes the type without running any conversion code. These are dangerous but sometimes necessary in low-level code:

- `unchecked_conversion` in Ada
- `reinterpret_cast` in C++

Type Compatibility

Most languages do not require type equivalence in every context.

Instead, the type of a value is required to be *compatible* with the context in which it is used.

What are some contexts in which type compatibility is relevant?

Type Compatibility

Most languages do not require type equivalence in every context.

Instead, the type of a value is required to be *compatible* with the context in which it is used.

What are some contexts in which type compatibility is relevant?

- *assignments*: type of lhs must be compatible with type of rhs
- *built-in functions* like $+$: operands must be compatible with integer or floating-point types
- *subroutine calls*: types of actual parameters (including return value) must be compatible with types of formal parameters

Type Compatibility

Definition of type compatibility varies greatly from language to language. Languages like ADA are very strict. Types are compatible if:

- they are equivalent
- they are both subtypes of a common base type
- both are arrays with the same number and types of elements in each dimension

Other languages, like C and FORTRAN are less strict. They automatically perform a number of type conversions.

An automatic, implicit conversion between types is called *type coercion*.

Coercion significantly weakens the security of the type system.

Type Coercion

Coercion in C

The following types can be freely mixed in C:

- `char`
- (`unsigned`) (`short`, `long`) `int`
- `float`, `double`

Recent trends in type coercion:

- *static typing*: stronger type system, less type coercion
- *user-defined*:
  - C++ allows user-defined type coercion rules
  - similar but cleaner: implicit conversions in Scala

Overloading and Coercion

Overloading:
Multiple definitions for the same name, distinguished by their types.
Overload resolution:
Process of determining which definition is meant in a given use.

- Usually restricted to functions
- Usually only for static type systems
- Related to coercion. Coercion can be simulated by overloading (but at a high cost). If type `a` has subtypes `b` and `c`, we can define three overloaded functions, one for each type. Simulation not practical for many subtypes or number of arguments.

Overload resolution based on:

- number of arguments
- argument types
- return type

Overloading and Coercion

What's wrong with this C++ code?

```
void f(int x);
void f(string *ps);

f(NULL);
```

Overloading and Coercion

What's wrong with this C++ code?

```
void f(int x);
void f(string *ps);

f(NULL);
```

Depending on how `NULL` is defined, this will either call the first function (if `NULL` is defined as `0`) or give a compile error (if `NULL` is defined as `((void*)0)`).

This is probably not what you want to happen, and there is no easy way to fix it. This is an example of *ambiguity* resulting from coercion combined with overloading.

Overloading and Coercion

What's wrong with this C++ code?

```
void f(int x);
void f(string *ps);

f(NULL);
```

Depending on how `NULL` is defined, this will either call the first function (if `NULL` is defined as `0`) or give a compile error (if `NULL` is defined as `((void*)0)`).

This is probably not what you want to happen, and there is no easy way to fix it. This is an example of *ambiguity* resulting from coercion combined with overloading.

There are other ways to generate ambiguity:

```
void f(int);
void f(char);
double d = 6.02;
f(d);
```

Generic Reference Types

An object of *generic reference type* can be assigned an object of any reference type.

- `void *` in C and C++
- `Object` in Java
- `AnyRef` in Scala

How do you go back to a more specific reference type from a generic reference type?

Generic Reference Types

An object of *generic reference type* can be assigned an object of any reference type.

- `void *` in C and C++
- `Object` in Java
- `AnyRef` in Scala

How do you go back to a more specific reference type from a generic reference type?

- Use a type cast
- Some languages include a tag indicating the type of an object as part of the object representation (Java, Scala, C#, C++)
- Others (such as C) simply have to settle for unchecked type conversions

Type Inference

How do you determine the type of an arbitrary expression?

Type Inference

How do you determine the type of an arbitrary expression?

Most of the time it's easy:

- the result of built-in operators (i.e. arithmetic) usually have the same type as their operands
- the result of a comparison is Boolean
- the result of a function call is the return type declared for that function
- an assignment has the same type as its left-hand side

Type Inference

How do you determine the type of an arbitrary expression?

Most of the time it's easy:

- the result of built-in operators (i.e. arithmetic) usually have the same type as their operands
- the result of a comparison is Boolean
- the result of a function call is the return type declared for that function
- an assignment has the same type as its left-hand side

Some cases are not so easy:

- operations on subranges
- operations on composite types

Type Inference for Subrange Operations

Consider this code:

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

What is the type of `a + b`?

Type Inference for Subrange Operations

Consider this code:

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

What is the type of `a + b`?

- Cheap and easy answer: base type of subrange, integer in this case
- More sophisticated: use bounds analysis to get `10..40`

Type Inference for Subrange Operations

Consider this code:

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

What if we assign to `a` an arbitrary integer expression?

Type Inference for Subrange Operations

Consider this code:

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

What if we assign to `a` an arbitrary integer expression?

- Bounds analysis might reveal it's OK (i.e. `(a + b) / 2`)
- However, in many cases, a run-time check will be required
- Assigning to some composite types (arrays, sets) requires similar run-time checks (e.g., `ArrayStoreException` in Java)

Records

A *record* consists of a set of typed fields.

Choices:

- *Name or structural equivalence?*
  Most statically typed languages choose name equivalence
  ML, HASKELL are exceptions
- *Nested records allowed?*
  Usually, yes. In FORTRAN and LISP, records but not record declarations can be nested
- *Does order of fields matter?*
  Typically, yes, but not in ML
- *Any subtyping relationship with other record types?*
  Most statically typed languages say no.
  Dynamically typed languages implicitly say yes.
  This is known as *duck typing*:

  If it walks like a duck and quacks like a duck, I call it a duck.
  -James Whitcomb Riley

Records: Syntax

Pascal:

```
type element = record
  name : array[1..2] of char;
  atomic_number : integer;
  atomic_weight : real;
end;
```

C:

```
struct element {
  char name[2];
  int atomic_number;
  double atomic_weight;
};
```

ML:

```
type element = {
  name: string,
  atomic_number: int,
  atomic_weight: real }
```

Records: Memory Layout

The order and layout of record fields in memory are tied to implementation trade-offs:

- *Alignment* of fields on memory word boundaries makes access faster, but may introduce *holes* that waste space.
- If holes are forced to contain zeroes, comparison of records is easier, but zeroing out holes requires extra code to be executed when the record is created.
- Changing the order of fields may result in better performance, but predictable order is necessary for some systems code.

Variant Records

A *variant record* is a record that provides multiple alternative sets of fields, only one of which is valid at any given time.

Each set of fields is known as a *variant*.

Because only one variant is in use at a time, the variants can share space.

In some languages (e.g. Ada, Pascal) a separate field of the record keeps track of which variant is valid.

In this case, the record is called a *discriminated union* and the field tracking the variant is called the *tag* or *discriminant*.

Without such a tag, the variant record is called a *nondiscriminated union*.

Variant Records in Ada

Need to treat group of related representations as a single type:

```
type Figure_Kind is (Circle, Square, Line);
type Figure (Kind: Figure_Kind := Square) is
record
  Color: Color_Type;
  Visible: Boolean;
  case Kind is
    when Line   => Length: Integer;
                   Orientation: Float;
                   Start: Point;
    when Square => Lower_Left, Upper_Right: Point;
    when Circle => Radius: Integer;
                   Center: Point;
  end case;
end record;
```

Variant Records in Ada

```
C1: Figure(Circle); -- discriminant cannot change
S1: Figure;         -- discriminant can change
...
C1.Radius := 15;
if S1.Lower_Left = C1.Center then ...

function Area (F: Figure) return Float is
  -- applies to any figure, i.e., subtype
begin
  case F.Kind is
    when Circle => return Pi * Radius ** 2;
  ...
end Area;
```

Variant Records in Ada

```
L : Figure(Line);
S : Figure(Square);
C : Figure;           -- defaults to Square
P1 := Point;
...
C := (Circle, Red, False, 10, P1);
   -- record aggregate: C is now a Circle
... C.Orientation ...
   -- illegal, circles have no orientation
C := L;
   -- C is now a line
S := L;
   -- illegal, S cannot change from Square
C.Kind := Square;
   -- illegal, discriminant can only be
   -- changed by assigning whole record
```

Nondiscriminated Unions

Nondiscriminated or *free* unions can be used to bypass the type model:

```
union value {
  char *s;
  int i;     // s and i allocated at same address
};
```

Keeping track of current type is programmer's responsibility.
Can use an explicit tag if desired:

```
struct entry {
  int discr;
  union {    // anonymous component, either s or i
    char *s; // if discr = 0
    int i;   // if discr = 1, but system won't che
  };
};
```

Note: no language support for safe use of variant!

Discriminated Unions and Dynamic Typing

In dynamically-typed languages, only values have types, not names.

```
(define S 13.45)    ; a floating-point number
...
(define S '(1 2 3 4)) ; now it's a list
```

Run-time values are described by discriminated unions.
Discriminant denotes type of value.

Arrays

An *array* is a mapping from an *index type* to an *element* or *component type*.

- *index types*: most languages restrict to an integral type
  Ada, Pascal, Haskell allow any discrete type
  most scripting languages allow non-discrete types (assoc. arrays)
- *index bounds*: many languages restrict lower bound:
  C, Java: 0, Fortran: 1, Ada, Pascal: no restriction
- *length*: when is it determined?
  Fortran: compile time; most other languages: can choose
- *dimensions:*
  Fortran has true multi-dimensional arrays
  most other languages simulate them quite nicely as arrays of arrays
- *first-classness*: in C/C++ functions cannot return arrays
- a *slice* or *section* is a rectangular portion of an array
  Some languages (e.g. Fortran, Perl, Python, Apl) have a rich set of array operations for creating and manipulating sections.

Array Literals

- Ada: `(23, 76, 14)`
- Scheme: `#(23, 76, 14)`
- C and C++ have initializers, but not full-fledged literals:

```
int v2[] = { 1, 2, 3, 4 };   // size from initializer

char v3[2] = { 'a', 'z'};    // declared size

int v5[10] = { -1 };         // default: other components = 

struct School r =
  { "NYU", 10012 };          // record initializer

char name[] = "Scott";       // string literal
```

Array Shape

The *shape* of an array consists of the number of dimensions and the bounds of each dimension in the array.

The time at which the shape of an array is bound has an impact on how the array is stored in memory:

- *global lifetime, static shape*: static global memory
- *local lifetime, static shape*: part of local stack frame
- *local lifetime, shape bound at runtime*: variable-size part of local stack frame
- *arbitrary lifetime, shape bound at runtime*: allocate from heap or reference to existing array
- *arbitrary lifetime, dynamic shape*: also known as *dynamic arrays*, must allocate (and potentially reallocate) in heap

Array Memory Layout

Two-dimensional arrays

- *Row-major layout*: Each row of array is in a contiguous chunk of memory
- *Column-major layout*: Each column of array is in a contiguous chunk of memory
- *Row-pointer layout*: An array of pointers to rows lying anywhere in memory

If an array is traversed differently from how it is laid out, this can dramatically affect performance (primarily because of cache misses)

A *dope vector* contains the dimension, bounds, and size information for an array. Dynamic arrays require that the dope vector be held in memory during run-time.

Pointers

- *value model* pointer has a value that denotes a memory location (C, PASCAL, ADA)
- *reference model* names have dynamic bindings to objects, pointer is implicit (ML, LISP, SCHEME, SCALA)
- JAVA uses value model for built-in (scalar) types, reference model for user-defined types

```
type Ptr is access Integer; -- Ada: named type

typedef int* ptr;  // C, C++
```

Extra pointer capabilities

Questions:

- Is it possible to get the address of a variable?
  Convenient, but aliasing causes optimization difficulties. (the same way that pass by reference does)
  Unsafe if we can get the address of a stack allocated variable (allowed in C/C++)
- Is pointer arithmetic allowed?
  unsafe if unrestricted
  In C/C++, no bounds checking:

```
// allocate space for 10 ints
int* p = malloc(10 * sizeof(int));
p += 42;
... *p ...  // out of bounds, but no check
```

Recursive data structures in C++

```
struct cell {
  int value;
  cell* prev;
  cell* next; // legal to mention name
};            // before end of declaration
struct list;  // incomplete declaration
struct link {
  link* succ;
  list* memberOf;
};            // a pointer to it
struct list { // full definition
  link* head; // mutual references
};
```

Pointers and dereferencing

- Need notation to distinguish pointer from designated object
  - in ADA: `Ptr` vs `Ptr.all`
  - in C: `ptr` vs `*ptr`
  - in JAVA: no notion of pointer (only references)
- For pointers to composite values, dereference can be implicit:
  - in ADA: `C1.Value` equivalent to `C1.all.Value`
  - in C/C++: `c1.value` and `c1->value` are different

Pointers and arrays in C/C++

In C/C++, the notions:

- an array
- a pointer to the first element of an array

are almost the same.

```
void f (int *p) { ... }
int a[10];
f(a);  // same as f(&a[0])

int *p = new int[4];
... p[0] ...  // first element
... *p ...    // first element
... p[1] ...  // second element
... *(p+1) ...// second element

... p[10] ... // past the end; undetected error
```

Pointers and safety

Pointers create aliases: accessing the value through one name affects retrieval through the other:

```
int *p1, *p2;
...
p1 = new int[10];  // allocate
p2 = p1;           // share
delete[] p1;       // discard storage
p2[5] = ...        // error:
                   //   p2 is dangling
```

Pointer troubles

Several possible problems with low-level pointer manipulation:

- dangling references
- garbage (forgetting to free memory)
- freeing dynamically allocated memory twice
- freeing memory that was not dynamically allocated
- reading/writing outside object pointed to

Dangling references

If we can point to local storage, we can create a reference to an undefined value:

```
int *f () {         // returns a pointer to an int
  int local;        // variable on stack frame of f
  ...
  return &local; // reference to local entity
}

int *x = f ();
...
*x = 5; // stack may have been overwritten
```

Lists, sets, and maps

- list: ordered collection of elements
- set: collection of elements with fast searching
- map: collection of (key, value) pairs with fast key lookup

Low-level languages typically do not provide these. High-level and scripting languages do, some as part of a library.

- PERL, PYTHON: built-in, lists and arrays merged.
- C, FORTRAN, COBOL: no
- C++: part of STL: `list<T>`, `set<T>`, `map<K,V>`
- JAVA, SCALA: yes, in standard library
- SETL: built-in
- ML, HASKELL: lists built-in, set, map part of standard library
- SCHEME: lists built-in
- PASCAL: built-in sets
  but only for discrete types with few elements, e.g., 32

Function types

- not needed unless the language allows functions to be passed as arguments or returned
- variable number of arguments:
  C/C++: allowed, type system loophole
  JAVA: allowed, but no loophole
- optional arguments: normally not part of the type.
- missing arguments in call: with dynamic typing typically OK.