CSE 307: Principles of Programming Languages

Spring 2015

R. Sekar

Topics

1. Simple/Built-in Types
2. Compound Types
3. Polymorphism
4. Type Equivalence
5. Type Compatibility
6. Type Checking

Section 1

Simple/Built-in Types

Simple Types

- Predefined
  - int, float, double, etc in C
  - int, bool, float, etc. in OCAML
- All other types are constructed, starting from predefined (aka primitive) types
  - Enumerated:
    - enum colors {red, green, blue} in C
    - type colors = Red|Green|Blue in OCAML
    - type is a keyword in OCAML to introduce new types

Section 2

Compound Types

Compound Types

- Types constructed from other types using type constructors
  - Cartesian product (*)
  - Function types ($\rightarrow$)
  - Union types ($\cup$)
  - Arrays
  - Pointers
  - Recursive types

Cartesian Product

- Let I represent the integer type and R represent real type.
- The cross product $I \times R$ is defined in the usual manner of product of sets, i.e.,

$$I \times R = \{(i, r) | i \in I, r \in R\}$$

Product Types (Continued)

- Product types correspond to "tuples" in OCAML.
- They are not supported in typical imperative languages, except with labels.
- Type on previous slide denoted int*float in OCAML.

```
# let v = (2,3.0);;
val v :  int * float = (2, 3.)
# type mytype = int * float;;
type mytype = int * float
```

- Note: type is a keyword to introduce new names (abbreviations) for types already known to OCAML, or for introducing new types unknown to OCAML.

Product Types (Continued)

- Cartesian product operator is non-associative:

```
# let t = (2,3,4.0);;
val t : int * int * float = (2, 3, 4.)
# let s = ((2,3), 4.0);;
val s : (int * int) * float = ((2, 3), 4.)
# let u = (2, (3,4.0));;
val u : int * (int * float) = (2, (3, 4.))
# t = s;;
Error: This expression has type (int * int) * float but an expression was
expected of type int * (int * float)
```

- Note: compiler complains that the types of arguments to equality operator must be the same, but it is not so in this case.
- You will get type error messages if you try to compare $s = u$ *or* $t = u$.

Product Types (Continued)

- Note: The equality operator has the type $'t *' t \rightarrow bool$ for any type t.
  - $'t$ is a type variable
  - Type variable names begin with a $'$
- Elements of a 2-tuple can be extracted using `fst` and `snd`:

```
# fst(u);;
- : int = 2
# snd(u);;
- : int * float = (3, 4.)
# snd(t);;
Error: This expression has type int * int * float but an expression
was expected of type 'a * 'b
# let third_of_four(_,_, x,_) = x;;
val third_of_four : 'a * 'b * 'c * 'd -> 'c = <fun>
```

- The error message says that `t` has more than two elements.

Labeled Product types

- In Cartesian products, components of tuples don't have names.
  - Instead, they are identified by numbers.
- In labeled products each component of a tuple is given a name.
- Labeled products are also called records (a language-neutral term)

Labeled Product types (Continued)

- `struct` is a term that is specific to C and C++

```
struct t {int a;float b;char *c;}; in C
type t = {a:int; b:float; c:string};; in OCAML
```

- In OCAML, components of a labeled tuple value can be accessed using the dot notation <identifier>.<field_name>

```
# type t = { a : int; b : float; c : string; };;
type t = { a : int; b : float; c : string; }
# let m = {a=1;b=2.0;c="abc"};;
val m : t = {a = 1; b = 2.; c = "abc"}
# m.c;;
- : string = "abc"
```

Function Types

- $T_1 \rightarrow T_2$ is a function type
  - Type of a function that takes one argument of type T_1 and returns type T_2
- OCAML supports functions as first class values.
  - They can be created and manipulated by other functions.
- In imperative languages such as C/C++, we can pass pointers to functions, but this does not offer the same level of flexibility.
  - E.g., no way for a C-function to dynamically create and return a pointer to a function;
  - rather, it can return a pointer to an EXISTING function

OCAML Examples of Function Types

- Example

```
# let f x = x * x;;
val f : int -> int = <fun>
# let g x y = x *. y;;
val g : float -> float -> float = <fun>
```

- Note: g is different from h given below.
  - g takes two arguments, which can be supplied one at a time
  - h takes only one argument, which is a tuple with two components.

```
# let h (x, y) = x *. y;;
val h : float * float -> float = <fun>
# let v = g 3.0;;
val v : float -> float = <fun>
```

Function Types (Continued)

- Type of g is float -> float -> float.
  - -> operator is right-associative, so we read the type as float -> (float -> float).
- When g is given one argument, it returns a new function value.
  - g, when given an argument of type float, returns a value of type (float -> float)

```
# let u = v 2.0;;
val u : float = 6.
```

- When a float argument is given to v, it consumes it and produces an output value of type float.
- v is called a “closure”
  - It represents a function for which some arguments have been provided, but its evaluation cannot proceed unless additional arguments are provided.
  - The closure "remembers" the arguments supplied so far

Union types

- Union types correspond to set unions, just like product types corresponded to Cartesian products.
  - -> operator is right-associative, so we read the type as float -> (float -> float).
- Unions can be tagged or untagged. C/C++ support only untagged unions:

```
union v {
  int ival;
  float fval;
  char cval;
};
```

Tagged Unions

- In untagged unions, there is no way to ensure that the component of the right type is always accessed.
  - E.g., an integer value may be stored in the above union, but due to a programming error, the fval field may be accessed at a later time.
  - fval doesn't contain a valid value now, so you get some garbage.
- With tagged unions, the compiler can perform checks at runtime to ensure that the right components are accessed.
- Tagged unions are NOT supported in C/C++.

Tagged Unions (Continued)

- Pascal supports tagged unions using VARIANT RECORDs

```
RECORD
  CASE b: BOOLEAN OF
    TRUE: i: INTEGER; |
    FALSE: r: REAL END
  END
END
```

- Tagged union is also called a discriminated union

Tagged Unions (Continued)

- Tagged unions are supported in OCAML using type declarations.

```
# type tt = Floatval of float | Intval of int;;
type tt = Floatval of float | Intval of int
# let v = Floatval (2.0);;
val v : tt = Floatval 2.
# let u = Intval (3);;
val u : tt = Intval 3
# let add (x, y) =
   match (x, y) with
      (Intval x1, Intval x2) -> Intval(x1+x2)
      | (Floatval x1, Floatval x2) -> Floatval(x1+.x2);;
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
(Floatval _, Intval _)
val add : tt * tt -> tt = <fun>
```

Tagged Unions (Continued)

- Tagged unions are supported in OCAML using type declarations.

```
# add (u, v);;
Exception: Match_failure ("//toplevel//", 14, 3).
# let w = Intval(3);;
val w : tt = Intval 3
# add(u,w);;
- : tt = Intval 6
```

- Note: we can redefine add as follows so as to permit addition of floats and ints.

```
# let add (x, y) =
   match (x, y) with
      (Intval x1, Intval x2) -> Intval(x1 + x2)
      | (Floatval x1, Floatval x2) -> Floatval(x1 +. x2)
      | (Intval x1, Floatval y1) -> Floatval(float_of_int(x1) +. y1)
      | (Floatval x1, Intval y1) -> Floatval(x1 +. float_of_int(y1));;
val add : tt * tt -> tt = <fun>
```

Array types

- Array construction is denoted by
  - array(<range>, <elememtType>).
- C-declaration
  - `int a[5];`
  - defines a variable a of type array(0-4, int)
- A declaration
  - `union tt b[6][7];`
  - declares a variable b of type array(0-4, array(0-6, union tt))
- We may not consider range as part of type

Pointer types

- A pointer type will be denoted using the syntax
  - ptr(<elementType>)
  - where <elementType> denote the types of the object pointed by a pointer type.
- The C-declaration
  - `char *s;`
  - defines a variable s of type ptr(char)
- A declaration
  - `int (*f)(int s, float v)`
  - defines a (function) pointer of type ptr(int*float $\rightarrow$ int)

Recursive types

- Recursive type: a type defined in terms of itself.
- Example in C:

```
struct IntList {
  int hd;
  intList tl;
};
```

- Does not work:
  - This definition corresponds to an infinite list.
  - There is no end, because there is no way to capture the case when the tail has the value “nil”

Recursive types (Continued)

- Need to express that tail can be nil or be a list.
- Try: variant records:

```
TYPE charlist = RECORD
   CASE IsEmpty:  BOOLEAN OF
     TRUE: /* empty list */ |
     FALSE:
       data:  CHAR;
       next:  charlist;
   END
END
```

- Still problematic: Cannot predict amount of storage needed.

Recursive types (Continued)

- Solution in typical imperative languages:
- Use pointer types to implement recursive type:

```
struct IntList {
  int hd;
  IntList *tl;
};
```

- Now, tl can be:
  - a NULL pointer (i.e., nil or empty list)
  - or point to a nonempty list value
- Now, IntList structure occupies only a fixed amount of storage

Recursive types In OCAML

- Direct definition of recursive types is supported in OCAML using type declarations.
- Use pointer types to implement recursive type:

```
# type intBtree =
   LEAF of int
   | NODE of int * intBtree * intBtree;;
type intBtree = LEAF of int | NODE of int * intBtree * intBtree
```

- We are defining a binary tree type inductively:
  - Base case: a binary tree with one node, called a LEAF
  - Induction case: construct a binary tree by constructing a new node that sores an integer value, and has two other binary trees as children

Recursive types In OCAML (Continued)

- We may construct values of this type as follows:

```
# let l = LEAF(1);;
val l : intBtree = LEAF 1
# let r = LEAF(3);;
val r : intBtree = LEAF 3
# let n = NODE(2, l, r);;
val n : intBtree = NODE (2, LEAF 1, LEAF 3)
```

Recursive types In OCAML (Continued)

- Types can be mutually recursive. Consider:

```
# type expr = PLUS of expr * expr
              | PROD of expr * expr
              | FUN  of (string * exprs)
              | IVAL of int
          and
          exprs= EMPTY |
                 LIST of expr * exprs;;
type expr =
    PLUS of expr * expr
  | PROD of expr * expr
  | FUN of (string * exprs)
  | IVAL of int
and exprs = EMPTY | LIST of expr * exprs
```

- The key word “and” is used for mutually recursive type definitions.

Recursive types In OCAML (Continued)

- We could also have defined expressions using the predefined list type:

```
#  type expr=PLUS of expr*expr
              | PROD of expr*expr
              | FUN  of string * expr list;;
type expr =
    PLUS of expr * expr
  | PROD of expr * expr
  | FUN of string * expr list
```

- Examples: The expression “3 + (4 * 5)” can be represented as a value of the above type expr as follows

Recursive types In OCAML (Continued)

- The following picture illustrates the structure of the value “pl” and how it is constructed from other values.

```
Pl ------>   PLUS
            /    \
           /      \
v3 ---> IVAL     PROD <----- pr
         |         /\
         |        /  \
         3   /->IVAL IVAL <--- v4
            /      |    |
         v5        |    |
                   5    4
```

```
let v3 = IVAL(3);;
let v5 = IVAL(5);;
let v4 = IVAL(4);;
let pr = PROD(v5, v4);;
let pl = PLUS(v3, pr);;
```

Recursive types In OCAML (Continued)

- Similarly, "f(2,4,1)" can be represented as:

```
let a1 = EMPTY;;
let a2 = ARG(IVAL(4), a1);;
let a3 = ARG(IVAL(2), a2);;
let fv = FUN("f", a3);;
```

- Note the use of "expr list" to refer to a list that consists of elements of type "expr"

Section 3

Polymorphism

Polymorphism

- Ability of a function to take arguments of multiple types.
- The primary use of polymorphism is code reuse.
- Functions that call polymorphic functions can use the same piece of code to operate on different types of data.

Overloading (adhoc polymorphism)

- Same function NAME used to represent different functions
  - implementations may be different
  - arguments may have different types
- Example:
  - operator '+' is overloaded in most languages so that they can be used to add integers or floats.
  - But implementation of integer addition differs from float addition.
  - Arguments for integer addition or ints, for float addition, they are floats.
- Any function name can be overloaded in C++, but not in C.
- All virtual functions are in fact overloaded functions.

Polymorphism & Overloading

- Parametric polymorphism:
  - same function works for arguments of different types
  - same code is reused for arguments of different types.
  - allows reuse of “client” code (i.e., code that calls a polymorphic function) as well
- Overloading:
  - due to differences in implementation of overloaded functions, there is no code reuse in their implementation
  - but client code is reused

Parametric polymorphism in C++

- Example:

```
template <class C>
Type min(const C* a, int size, C minval) {
  for (int i = 0; i < size; i++)
    if (a[i] < minval)
      minval = a[i];
  return minval;
}
```

- Note: same code used for arrays of any type.
  - The only requirement is that the type support the "<" and "=" operations
- The above function is parameterized wrt class C
  - Hence the term "parametric polymorphism".
- Unlike C++, C does not support templates.

Code reuse with Parametric Polymorphism

- With parametric polymorphism, same function body reused with different types.
- Basic property:
  - does not need to "look below" a certain level
  - E.g., min function above did not need to look inside each array element.
  - Similarly, one can think of length and append functions that operate on linked lists of all types, without looking at element type.

Code reuse with overloading

- No reuse of the overloaded function
  - there is a different function body corresponding to each argument type.
- But client code that calls a overloaded function can be reused.
- Example
  - Let C be a class, with subclasses C1,...,Cn.
  - Let f be a virtual method of class C
  - We can now write client code that can apply the function f uniformly to elements of an array, each of which is a pointer to an object of type C1,...,Cn.

Example

- Example:

```
void g(int size, C *a[]) {
  for (int i = 0; i < size; i++)
    a[i]->f(...);
}
```

- Now, the body of function g (which is a client of the function f) can be reused for arrays that contain objects of type C_1 or C_2 or ... or C_n,or even a mixture of these types.

Parameterized Types

- Type declarations for parameterized data types:

```
type (<typeParameters>) <typeName> = <typeExpression>
type ('a, 'b) pairList = ('a * 'b) list;;
```

- Define Btree:

```
# type ('a,'b) btree = LEAF of 'a
                     | NODE of 'b * ('a,'b) btree * ('a,'b) btree;;
type ('a, 'b) btree =
        LEAF of 'a
      | NODE of 'b * ('a, 'b) btree * ('a, 'b) btree
# type intBTree = (int, int) btree;;
type intBTree = (int, int) btree
```

Example Functions and their Type

```
# let rec leftmost(x) =
      match x with
         LEAF(x1) -> x1
         | NODE(y, l, r) -> leftmost(l);;
val leftmost : ('a, 'b) btree -> 'a = <fun>

# let rec discriminants(x) =
   match x with
      LEAF(x1) -> []
      | NODE(y,l,r) -> let l1 = discriminants(l)
                             in let l2 = discriminants(r) in l1@(y::l2);;
val discriminants : ('a list, 'b) btree -> 'b list = <fun>
```

Example Functions (Continued)

```
# let rec append(x,y) =
      match x with
         x1::xs -> x1::append(xs,y)
         | [] -> y;;
val append : 'a list * 'a list -> 'a list = <fun>

# let rec f(x,y) =
      match x with
         x1::xs -> x1::f(xs,y)
         | [] -> [];;
val f : 'a list * 'b -> 'a list = <fun>
```

- OCAML Operators that restrict polymorphism:
  - Arithmetic, relational, boolean, string, type conversion operators
- OCAML Operators that allow polymorphism
  - tuple, projection, list, equality (= and <>)

Section 4

Type Equivalence

Type Equivalence

- Structural equivalence: two types are equivalent if they are defined by identical type expressions.
  - array ranges usually not considered as part of the type
  - record labels are considered part of the type.
- Name equivalence: two types are equal if they have the same name.
- Declaration equivalence: two types are equivalent if their declarations lead back to the same original type expression by a series of redeclarations.

Type Equivalence (contd.)

- Structural equivalence is the least restrictive
- Name equivalence is the most restrictive.
- Declaration equivalence is in between
- TYPE t1 = ARRAY [1..10] of INTEGER; VAR v1: ARRAY [1..10] OF INTEGER;
- TYPE t2 = t1; VAR v3,v4: t1; VAR v2: ARRAY [1..10] OF INTEGER;

	Structurally equivalent?	Declaration equivalent?	Name equivalent?
t1,t2	Yes	Yes	No
v1,v2	Yes	No	No
v3,v4	Yes	Yes	Yes

Declaration equivalence

- In Pascal, Modula use decl equivalence
- In C
  - Declequivusedforstructsandunions
  - Structualequivalenceforothertypes.

```
struct { int a ; float b ;} x ;
struct { int a; float b; }y;
```

- x and y are structure equivalent but not declaration equivalent.

```
typedef int* intp ;
typedef int** intpp ;
intpp v1 ;
intp *v2 ;
```

- v1 and v2 are structure equivalent.

Section 5

Type Compatibility

Type Compatibility

- Weaker notion than type equivalence
- Notion of compatibility differs across operators
- Example: assignment operator:
  - v = expr is OK if <expr> is type-compatible with v.
  - If the type of expr is a Subtype of the type of v, then there is compatibility.
- Other examples:
  - In most languages, assigning integer value to a float variable is permitted, since integer is a subtype of float.
  - In OO-languages such as Java, an object of a derived type can be assigned to an object of the base type.

Type Compatibility (Continued)

- Procedure parameter passing uses the same notion of compatibility as assignment
  - Note: procedure call is a 2-step process
    - assignment of actual parameter expressions to the formal parameters of the procedure
    - execution of the procedure body
- Formal parameters are the parameter names that appear in the function declaration.
- Actual parameters are the expressions that appear at the point of function call.

Section 6

Type Checking

Type Checking

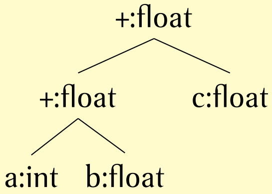
- Static (compile time)
  - Benefits
    - no run-time overhead
    - programs safer/more robust
- Dynamic (run-time)
  - Disadvantages
    - runtime overhead for maintaining type info at runtime
    - performing type checks at runtime
  - Benefits
    - more flexible/more expressive

Examples of Static and Dynamic Type Checking

- C++ allows
  - casting of subclass to superclass (always type-safe)
  - superclass to subclass (not necessarily type-safe) âĂŞ but no way to check since C++ is statically typed.
- Java uses combination of static and dynamic type-checking to catch unsafe casts (and array accesses) at runtime.

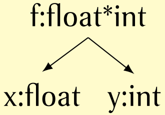
Type Checking (Continued)

- Type checking relies on type compatibility and type inference rules.
- Type inference rules are used to infer types of expressions. e.g., type of (a+b)+c is inferred from type of a, b and c and the inference rule for operator '+'.
- Type inference rules typically operate on a bottom-up fashion.
- Example: (a+b)+c



Type Checking (Continued)

- In OCAML, type inference rules capture bottom-up *and* top-down flow of type info.
- Example of Top-down: let f x y:float*int = (x, y)



- Here types of x and y inferred from return type of f.
- Note: Most of the time OCAML programs don't require type declaration.
  - But it really helps to include them: programs are more readable, and most important, you get far fewer hard-to-interpret type error messages.

Strong Vs Weak Typing

- Strongly typed language: such languages will execute without producing uncaught type errors at runtime.
  - no invalid memory access
    - no seg fault
    - array index out of range
    - access of null pointer
  - No invalid type casts
- Weakly typed: uncaught type errors can lead to undefined behavior at runtime
- In practice, these terms used in a relative sense
- Strong typing does not imply static typing

Type Conversion

- Explicit: Functions are used to perform conversion.
  - example: strtol, atoi, itoa in C; float and int etc.
- Implicit conversion (coercion)
  - example:
    - If a is float and b is int then type of a+b is float
    - Before doing the addition, b must be converted to a float value. This conversion is done automatically.
- Casting (as in C)
- Invisible “conversion:” in untagged unions

Data Types Summary

- Simple/built-in types
- Compound types (and their type expressions)
  - Product, union, recursive, array, pointer
- Parametric Vs subtype polymorphism, Code reuse
- Polymorphism in OCAML, C++,
- Type equivalence
  - Name, structure and declaration equivalence
- Type compatibility
- Type inference, type-checking, type-coercion
- Strong Vs Weak, Static Vs Dynamic typing