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

- Types constructed from other types using type constructors
  - Cartesian product (*)
  - Function types (→)
  - Union types (∪)
  - Arrays
  - Pointers
  - Recursive types
Simple/Built-in Types  Compound Types  Polymorphism  Type Equivalence  Type Compatibility

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.
  ```ocaml
  # let v = (2,3.0);;
  val v : int * float = (2, 3.0)
  # 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:
  ```ocaml
  # 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 \( t \times t \rightarrow bool \) for any type \( t \).
  - \( t \) is a type variable
  - Type variable names begin with a ‘
- Elements of a 2-tuple can be extracted using \texttt{fst} and \texttt{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)

- \texttt{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 \texttt{<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

```ocaml
# 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.

```ocaml
# 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.
  - $\rightarrow$ 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)
    ```ocaml
    # 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:

```c
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

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

- Tagged union is also called a discriminated union
Tagged Unions (Continued)

- Tagged unions are supported in OCAML using type declarations.

```ocaml
# 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>
```

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

```ocaml
# 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>
```

Tagged Unions (Continued)

- Array construction is denoted by
  - `array(range, elementType)`.

- 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
  
  \[
  \text{ptr(<element\text{Type}>)}
  \]
  
  where \(<\text{element\text{Type}}>\) denote the types of the object pointed by a pointer type.

- The C-declaration
  
  \[
  \text{char } *s;
  \]
  
  defines a variable \(s\) of type \(\text{ptr(char)}\)

- A declaration
  
  \[
  \text{int } (*f)(\text{int } s, \text{ float } v)
  \]
  
  defines a (function) pointer of type \(\text{ptr(int\text{*float } \rightarrow \text{ int})}\)

Recursive types

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

- Example in C:
  
  \[
  \text{struct IntList} \\
  \text{ int } hd; \\
  \text{ IntList tl;}
  \]

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

- Still problematic: Cannot predict amount of storage needed.
Recursive types (Continued)

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

```c
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:

```ocaml
# 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 stores 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:

```ocaml
# 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)
```
Types can be mutually recursive. Consider:

```ocaml
# 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.

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

```ocaml
# 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:

The following picture illustrates the structure of the value “pl” and how it is constructed from other values.
Similarly, “f(2,4,1)” can be represented as:

```ocaml
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”
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:
  ```cpp
  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:
  ```c
  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 C₁ or C₂ or ... or Cₙ, or even a mixture of these types.
Parameterized Types

- Type declarations for parameterized data types:
  
  \[
  \text{type } (<\text{typeParameters}>) <\text{typeName}> = <\text{typeExpression}> \\
  \text{type } ('a, 'b) \text{ pairList} = ('a * 'b) \text{ list};
  \]

- Define Btree:
  
  \[
  \begin{align*}
  \text{# type } ('a, 'b) \text{ btree} & = \text{ LEAF of 'a} \\
  & \quad | \text{ NODE of 'b * ('a, 'b) btree * ('a, 'b) btree};
  \end{align*}
  \]

Example Functions and their Type

- 
  \[
  \begin{align*}
  \text{# let rec leftmost(x) =} \\
  & \text{ match x with} \\
  & \quad \text{ LEAF(x1) } -> x1 \\
  & \quad \quad | \text{ NODE(y, 1, r) } -> \text{ leftmost(l);}
  \end{align*}
  \]

- 
  \[
  \begin{align*}
  \text{val leftmost : ('a, 'b) btree } & -> 'a = <\text{fun}> \\
  \end{align*}
  \]

Example Functions (Continued)

- 
  \[
  \begin{align*}
  \text{# let rec discriminants(x) =} \\
  & \text{ match x with} \\
  & \quad \text{ LEAF(x1) } -> [] \\
  & \quad \quad | \text{ NODE(y, 1, r) } -> \text{ let l1 = discriminants(l)} \\
  & \quad \text{ in let l2 = discriminants(r) in l1@[y::l2];}
  \end{align*}
  \]

- 
  \[
  \begin{align*}
  \text{val discriminants : ('a list, 'b) btree } & -> 'b list = <\text{fun}> \\
  \end{align*}
  \]

OCAML Operators that restrict polymorphism:

- Arithmetic, relational, boolean, string, type conversion operators

OCAML Operators that allow polymorphism

- tuple, projection, list, equality (= and <>)

41 / 57
Section 4

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

<table>
<thead>
<tr>
<th></th>
<th>Structurally equivalent?</th>
<th>Declaration equivalent?</th>
<th>Name equivalent?</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1,t2</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>v1,v2</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>v3,v4</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Declaration equivalence

- In Pascal, Modula use decl equivalence
- In C
  - Declequiv used for structs and unions
  - Structualequivalence for other types.
  ```c
  struct { int a; float b; } x;
  struct { int a; float b; } y;
  ```
  - x and y are structure equivalent but not declaration equivalent.
- v1 and v2 are structure equivalent.

Section 5

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**

- **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) 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: \((a+b)+c\)

\[
+ : \text{float} \\
+ : \text{float} \quad c : \text{float} \\
\text{a : int} \quad \text{b : float}
\]

Type Checking (Continued)

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

\[
f : \text{float*int} \\
f \quad x: \text{float} \quad y: \text{int}
\]

- 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