CS131: Programming Languages

Lun Liu

04.21.17

Type Alias

- OCaml allows names to be introduced as abbreviations of types (type alias)

```
type t = te
```

```
# type vector = int list;;
type vector = int list
# (fun (x : vector) -> x) [1;2;3];;
- : vector = [1; 2; 3]
```

User-Defined Types - Variant Types

- OCaml allows creating new types by defining a set of constructors

```
type t = C1 [of te1] | . . . | Cn [of ten]
```

- `C1, C2, …, Cn` are `constructors`
- Can use `constructors` to contruct values of type *t*
- `C1: te1 -> t`
- `C2: te2 -> t`
- `...`

User-Defined Types - Variant Types

```
# type sign = Positive | Zero | Negative;;
type sign = Positive | Zero | Negative
# Positive;;
- : sign = Positive
```

- In C:

```
enum sign { positive, zero, negative };
```

User-Defined Types - Variant Types

```
# type binary_tree =
          | Leaf of int
          | Tree of binary_tree * binary_tree;;
type binary_tree = Leaf of int | Tree of
binary_tree * binary_tree
# Tree (Leaf 3, Leaf 4);;
- : binary_tree = Tree (Leaf 3, Leaf 4)
```

User-Defined Types – Parameterized Variants

- User-defined types can be polymorphic

```
# type 'a binary_tree =
    | Leaf of 'a
    | Tree of 'a binary_tree * 'a binary_tree;;
type 'a binary_tree = Leaf of 'a | Tree of 'a
binary_tree * 'a binary_tree
```

Pattern Matching on User-Defined Types

```
let f x = match x with
      C1(a1, …) -> e1
     |C2(a2, …) -> e2
     …
     |Cn(an, …) -> en
```

Pattern Matching on User-Defined Types

```
# let rec size t = match t with
   | Leaf a -> 1
   | Tree(l, r) -> 1 + (size l) + (size r);;
val size : 'a binary_tree -> int = <fun>
# size (Tree (Tree (Leaf 3, Leaf 4), Leaf 5));;
- : int = 5
```

Exercises

- Peano Arithmetic
- Trees

Scoping

- Which *variable declaration* does a particular *variable usage* refer to? (Name Resolution)
- Static Scoping (Lexical Scoping)
  - Depends on the location in the source code and the **lexical context**, which is defined by where the named variable or function is defined
- Dynamic Scoping
  - Depends upon the program state when the name is encountered which is determined by the *execution context* or *calling context*.

https://en.wikipedia.org/wiki/Scope_(computer_science)

Scoping

Static scoping

```
int b = 5;
int foo()
{
    int a = b + 5;
    return a;
}

int bar()
{
    int b = 2;
    return foo();
}

int main()
{
    foo();
    bar();
    return 0;
}
```

Dynamic scoping

```
int b = 5;
int foo()
{
    int a = b + 5;
    return a;
}

int bar()
{
    int b = 2;
    return foo();
}

int main()
{
    foo();
    bar();
    return 0;
}
```

https://msujaws.wordpress.com/2011/05/03/static-vs-dynamic-scoping/

Scoping

Static scoping

```c
int b = 5;
int foo()
{
    int a = b + 5;
    return a;
}

int bar()
{
    int b = 2;
    return foo();
}

int main()
{
    foo(); // returns 10
    bar(); // returns 10
    return 0;
}
```

Dynamic scoping

```c
int b = 5;
int foo()
{
    int a = b + 5;
    return a;
}

int bar()
{
    int b = 2;
    return foo();
}

int main()
{
    foo(); // returns 10
    bar(); // returns 7
    return 0;
}
```

https://msujaws.wordpress.com/2011/05/03/static-vs-dynamic-scoping/

Type Checking

- Why do we need type systems?
  - FOR SAFETY!!
- Static Type Checking
  - Compile Time
- Dynamic Type Checking
  - Execution Time

Type Checking

- Static Type Checking
  - Early error detections
  - Guarantees for all possible executions
  - Documentation
  - Efficiency
  - Enforce constraints of user-defined types
- Dynamic Type Checking
  - More flexible
  - Quick development
  - Relative concise code

Backup