Polymorphism

COMP 302

07 POLYMORPHISM

Polymorphism

The ability to use a function as though it has many different types is called polymorphism

For example:

```
swap (3, 7) (* type: int * int -> int * int  *)
swap (“foo”, “bar”)
swap (“foo”, 1.234)
But there is no swap of type
int * real -> string * int
```

Type Checking

If, in the definition, x is of type ‘a, the only operations that can be performed on x are those that could be performed on any arbitrary type ‘a

If x is of type ‘a and y of type ‘a, x+y would not be a permitted operation

Polymorphic DataTypes

We can define options for any type individually.

But we also are able to define datatypes which depend on type variables:

```
datatype ‘a option =
    NONE
  | SOME of ‘a

val x : int option = SOME 17
val y : string option = SOME “Nathan”
```

Parameterized Types

New type:

a’ option is a type for any type a’

Building:

- NONE has type ‘a option
- SOME e, has type a’ option if e has type a’

Polymorphic Lists

The built-in type list is also polymorphic.

That is, we can define lists whose values are of any type: int list, string list, int*int list (see assignment 1), (int list) list, etc

```
fun length (lst : ‘a list) : int =
    case lst of
      []  => 0
     | x :: xs => 1 + length xs
```

The type of length is `‘a list -> int`

Parameterized Type Definitions

Earlier we saw that we can define a recursive integer list type:

```
datatype intlist = Nil | Cons of int * intlist
```

Rather than define a list type for reals, strings, etc, we are also able to define a parameterized datatype:

```
datatype ‘a list = Nil | Cons of ‘a * ‘a list
```

a’ is a type parameter

Examples:

```
val lst1 : int list = Cons(1, Cons(2, Cons(3, Nil)))
val lst2 : real list = Cons(3.14, Cons(2.57, Nil))
val lst3 : string list = Cons(“foo”, Cons(“bar”, Nil))
```

Functions on Lists

We can use pattern matching to decompose user defined recursive datatypes.

Note that every function uses a case with the same pattern

```
fun length (lst : ‘a list) : int =
  case lst of
    Nil => 0
  | Cons (h, t) => 1 + length (t)

fun isEmpty (lst : ‘a list) : bool =
  case lst of
    Nil => true
  | Cons (_, _) => false
```

Functions on Lists

```
fun append (lst1 : ‘a list, lst2 : ‘a list) : ‘a list =
  case lst1 of
    Nil => lst2
  | Cons (h, t) => Cons (h, append (t, lst2)

fun reverse (lst : ‘a lst) : ‘a list =
  case lst of
    Nil => Nil
  | Cons (h, t) => append (reverse(t), Cons(h, Nil))
```

Type Inference

It is possible to leave off extraneous type descriptions in function definitions

The type inference system of SML will fill in the types

We explore this later in the course, but introduce some basic ideas here

```
fun length lst =
    case lst of
      []  => 0
     | x :: xs => 1 + length xs
```

Annotate

Step 1: SML will annotate the function with type variables

```
fun length (lst : ‘a) : ‘b =
    case lst of
      []  => 0
     | x :: xs => 1 + length xs
```

Constraints

Step 2: SML will generate and them solve constraints

Since lst is case-analyzed using nil and cons we generate the constraint:

- ‘a = ‘c list for some type ‘c.

Since the function evaluates to 0 for the null list, we generate the constraint:

- ‘b = int

The system is **underconstrained** so the type of the function is the polymorphic type

```
‘c list -> int
```

Example 2

```
fun add (x:int, y:int) : int = x + y
fun sum (lst) =
    case lst of
      []  => 0
     | x :: xs => add (x, sum xs)
```

Annotation

```
fun add (x:int, y:int) : int = x + y
fun sum (lst : ‘a) : ‘b =
    case lst of
      []  => 0
     | x :: xs => add (x, sum xs)
```

Constraints

By analyzing the case pattern we have ‘a = ‘c list

By analyzing the empty branch, we have ‘b = int

By applying the add function we get ‘c = int

Solving the constraints gives us ‘a= int list and ‘b = int

This function definition is fully constrained and is not polymorphic

One More

```
fun sum (lst : ‘a) : ‘b =
    case lst of
      []  => “Nathan”
     | x :: xs => add (x, sum xs)
```

By analyzing the case pattern we have ‘a = ‘c list

By analyzing the empty branch, we have ‘b = string

By applying the add function we get ‘b = int

This code is **overconstrained** or ill-typed