CIS 425 Lecture Notes 10.17.2019

Maxwell Logan and Taylor Scott

October 17th, 2019

Last Time:

Racket: Dynamically typed: Types are checked at run-time. This makes it easier to compile, but it's far more error prone during runtime.
Example of declaring a list:

```
(list 1 true 3 4.6)
```

Note: All elements of a list do not need to have the same type

ML: Statically-typed: Types are checked at compile-time. This does not necessarily mean that one must declare types when creating variables, expressions, functions, etc., since types can be inferred automatically. However, it becomes harder to get a program to compile, but on the other hand less debugging is required.
Example of declaring a list:

```
[1, 2, 3, 4]
```

Note: All elements of a list need to have the same type

Today:

Example of a valid list in Racket:

```
(list 2 true "a")
```

The same list is invalid in ML:

```
[2 true "a"]
```

This is because ML needs to have unique types in its expressions. Here is a valid list in ML:

```
[1, 2]
```

This list has the type "int list", as you may expect. The empty list, represented as:

```
"[]" or "nil"
```

has the type **'a list**, which is a **polymorphic** type, meaning it is a generalized type.

Cons: cons stands for constructor, and it is a keyword in both Racket and ML. Let's see how it is used in ML. The list

```
[1, 2]
```

can be represented in cons notation by:

```
cons (1, cons (2, nil));
```

Or in infix notation:

```
1::2::nil
```

But what is the typing of cons? Well, here, it takes an int and an int list, and returns an int list.

```
cons:  int * int list -> int list
```

What about the type of the following tuple?

```
(2, true)
```

This tuple, unlike the similar list, is valid in ML and holds two types, an int and a bool (Represented as "int * bool").

Function Typing:

```
fun f x => x + 1;
```

has the typing "$int \rightarrow int$ " because it is inferred that x is an integer based on the addition operation between x and 1. On the other hand,

```
fun f x = true;
```

has the typing "$\alpha \rightarrow bool$" since the variable x does not have an assigned type and does not interact with the boolean in the body of the function.

In this next example, the function h has the typing "$int \rightarrow int$".

```
fun h x = x + x;
```

This is because the addition operator is an overloaded symbol, which means it has multiple definitions. The default in ML is an integer. You can't operate on integers and floats in the same expression. There's also no implicit coercion in ML which means you can't implicitly change types. Here's another example of how types work in ML:

`[[1, 2], 3];` Wrong
`[[9], [1]];` Right

cons notation:
`(9::nil)::(1::nil)::nil`

For a multiplication: `calc(3+2);`
Evaluated as calc(5) `calc'(+ 3 2);`

`datatype E = Lit of int|Plus of E * E`
Lit and E are constructors. Lit = literal

```
((2 + 3)+ 1)
(+(+ 3 2) 1) (P(P(Lit 3),Lit 2),Lit 1) //type E
```

In this example, x must be an int and Lit x is the base case.

```
fun calc(Lit x) = x
 | calc(P(x, y)) = (calc (x))*(calc(y));
```

$E \rightarrow int$

calc is of type function
$E \to int$
Lit: $nil \to E$
P: $E * E \to E$

```
calc(P(Lit 1, Lit 5))
```

$x \mapsto Lit\ 1$
$y \mapsto Lit\ 5$

```
calc(Lit 1)*(calc(Lit 5))
          1 * 5
            5
```

Floats cannot be converted to ints:

```
fun ?  x = if x then 1 + 1
           else 1.0 + 1.0
```

```
datatype iList = empty | icons of int * ilist;
datatype bList = bempty | bcons of bool * blist;
datatype α List = nil | cons of α* α list;
```