CIS 425 Eager versus Lazy evaluation

Eager vs. Lazy Languages

Lets say we have this code. What happens when we run it?

```
fun loop x = loop x;
fun zero x = 0;
zero(loop(9));
```

This code loops forever in ML, and would loop forever in Javascript. This is because these kind of programming languages are ”eager”: they compute things, like the invocation of the loop function even if they might not need the result. Other languages, such as Haskell, are ”lazy,” waiting until computations are needed before triggering them. In a lazy language, this will not loop forever.

Because of this, lazy languages can work with infinite structures. Consider the structure below, which attempts to represent the (famously never-ending) natural numbers.

```
fun nats n = n :: nats (n+1)
nats 0;
```

Once again, this will run forever! So, how can we get ML (or any other eager language) to behave like a lazy language, and wait to generate the next natural number until we directly request it?

The answer, broadly, is through functions. Consider the delay function below:

```
val delay = fn() => 2+2;
```

At this point in time, we have not calculated $2+2$ yet. It does not get executed until we actually call the function.

```
delay(); (* Now we calculate 2+2 *)
```

We can utilize this concept i when defining our sequence of natural numbers, like this[1].

```
datatype int_Seq = S of int * (unit -> int_Seq);
fun nats x = S(x, fn() => nats  (x+1));

fun get 1 (S(x, promise)) = x
  | get n (S(x, promise)) = get  (n-1) (promise());
```

[1]Note that I am using a different name for the constructor, just to emphasize that the usee chooses the name. Also, the function `get` is the uncurried version of the `get` function given in class.

Using this method, we can represent infinite sequences in any eager language, not just ML.

References - L-values versus R-values

```
val a = ref 10;
val a = ref 10 : int ref
val b = ref true;
val b = ref true : bool ref
```

A reference is similar to a pointer in C. Like a pointer, a reference points to data allocated in the heap. Unlike in C, pointer arithmetic is not allowed:

```
b + 2;
Error: operator and operand do not agree
```

To dereference a reference use !. Using ! is the same as * in C++.

```
!;
val it = fn : 'a ref -> 'a
!a;
val it = 10 : int
!b;
val it = true : bool
(op :=);
val it = fn : 'a ref * 'a -> unit
a := a + 1;
Error
a:= !a +1 ;
val it = () : unit
```

Note that the assignment returns unit because that is the value of the side effect. The value variable `a` points to now to 11.

References in ML follow uniform representation. This means that when space is allocated for a reference, it will be the same size regardless of the type.

Note that `:=` takes a reference to `a` as the L-value and an `a` as the R-value, then returns a unit. This differs from `=` in that `:=` is updating a value and `=` is introducing a new definition.

Encoding of Objects

```
datatype Message = GetBalance $\vert$  Deposit of int $\vert$
    Withdraw of int;

fun opening_account init_amt  =
                     let
                      val amount = ref  init_amt;
                      in
                         fn Deposit     amt => (amount := !amount +
     amt; !amount)
                           | Withdraw   amt => (amount := !amount -
     amt;!amount)
```

```
                              | GetBalance      => !amount
                            end;

val account1 = opening_account 10;
val account2 = opening_account 90;
account1 (Withdraw 15);
account1 (Deposit 100);
account2 (Withdraw  20);
```

This is similar to an object in Python or Java. The val amount only needs to be allocated once. `account1` and `account2` now provide an interface to call the functions defined in opening_account the same way an object provides an interface to call methods.

Polymorphism vs. Overloading

In ML, the `+` operator is overloaded, and the `map` function is polymorphic. Both operators can work on different datatypes. What's the difference?

As we know, `+` can work on both ints and reals. However, it is not the same function at runtime. When the compiler sees a `+` sign, it determines whether it is acting on `ints` or on `reals`, and decides, before runtime, which addition function to call.

This is not the case for polymorphic functions. In this case, the compiler generates one piece of code, that will be used for any data type passed through it. However, different types have different sizes: a `real` often takes more bytes than an`int`, and custom types could be even larger. How does the compiler generate a single piece of machine code, if it doesn't know how many bytes to allocate for the function argument?

The solution to that is to exclusively pass by reference in a polymorphic function. All pointers are the same size, so the compiler only needs to allocate enough space for a pointer for the argument. This solution comes at a cost. Passing by reference is often less efficient than passing by value, which is a reason why many languages do not support polymorphic functions.

The code for `swap` illustrates the difference between polymorphism and overloading.

```
fun swap a b = let val temp = !a
                 in (a:= !b; b := temp)

                 end;

template <typename T>
void swap(T& x, T& y){
T tmp = x; .....   ;
}
```

The C++ version doesn't generate a single piece of code for each type. It generates unique code for each type. The C++ `swap` is overloaded because different code is executed at run time depending on the type. While the ML `swap` is polymorphic because the same code is generated independent of the types. This require a uniform representation, which in case of ML consists of a pointer. For example, the variable `temp` will correspond to a pointer to an `int`, in case we are swapping integers. Instead, the variable `tmp` in the C++ code will be allocated on the stack and it will occupy the space for an `int`.

Static vs Dynamic Typing

We summarize the advantages of static vs dynamic typing.

Static

- Efficiency
- Fewer tests needed: properties of your programs are already checked
- Better Documentation
- Safety net for maintenance

Dynamic

- Simpler languages
- Fewer puzzling compiler errors
- Easier for explorations
- No type-imposed limits to expressiveness