

Solutions to Homework 4

Chris Fensch & Chris Stork

September 14, 2004

This homework is worth 115 points, i.e., there are 15 extra points.

1 Polymorphism

Shortly describe the three different kinds of polymorphism that we learned about (or will learn about on Wednesday). Give a specific example for each. (10 Points)

Answer:

Parametric Polymorphism Parametrically polymorph functions are applicable to any arguments which match the function type expression (which can include type variables). Examples are our `map` and `append` functions.

Ad-Hoc Polymorphism Multiple implementations that operate on arguments of different type, but are invoked by the same name (same as overloading). Our standard example was the overloading of the plus operator in ML, which can either operate on integers or reals.

Subtype Polymorphism Expressions can be polymorphic in the sense that types are replaceable by subtypes. The birds exercise at the end should be an example of this, or look at A.1.2 in Mitchell for another one.

2 Type Inference

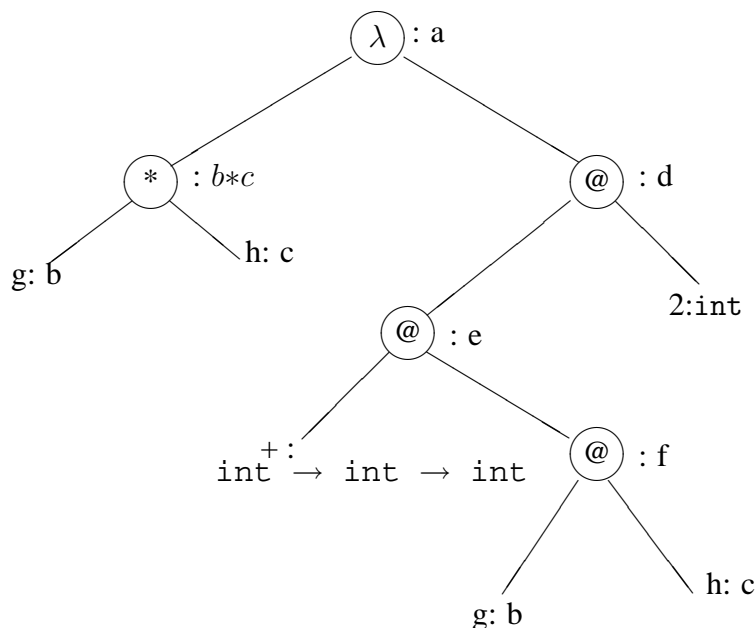
```
fun f(g,h) = g(h) + 2;
```

- (a) Annotate F 's AST below with types at each node.
- (b) Write down the type constraint equations.
- (c) Transcribe the equations into a form solvable by our `unify` Scheme function. Translate $x \rightarrow y$ and $x * y$ to the two argument relation (`arrow x y`) and (`tuple x y`), respectively, where x and y stand for arbitrary types/terms.
- (d) Confirm that our MGU corresponds to the actual type given by ML.

(20 Points)

Answer:

(a)



(b)

$$\begin{aligned}
 b &= c \rightarrow f \\
 \text{int} \rightarrow \text{int} \rightarrow \text{int} &= f \rightarrow e \\
 e &= \text{int} \rightarrow d \\
 a &= b * c \rightarrow d
 \end{aligned}$$

Note that some type constraint (i.e., equations) are implicit by our choice of type names in (a).

(c) Since `unify` operates on two terms, we first need a way to pack the four equations into two terms. The idea is that we use a fake function `f` that takes four arguments. In the first function we insert all the left hand sides of the type constraints equations; in the second all the right hand sides.

```

> (unify '(f ?b          (arrow int (arrow int int)) ?e          ?a)
      '(f (arrow ?c ?f) (arrow ?f ?e)          (arrow int ?d) (arrow (tuple ?b ?c) ?d)))
(((?b arrow ?c int) (?f . int) (?d . int) (?e arrow int int) (?a arrow (tuple (arrow ?c int) ?c) int)))

```

The final answer is given by `(?a arrow (tuple (arrow ?c int) ?c) int)`. Translating it back into infix notation we get:

$$a = ((c \rightarrow \text{int}) * c) \rightarrow \text{int}.$$

(d) ML gives us the following type for the function:

```
- fun f(g,h) = g(h) + 2;  
> val 'a f = fn : ('a -> int) * 'a -> int
```

This type is identical to the one we got. Just replace *c* with 'a.

3 Parametric Polymorphism in C++ and ML

- (a) *Hand in a corrected version of the swap function as given by Mitchell on page 149. Name the file swap.ml. (5 Points)*
- (b) *Shortly describe the consequences of the different realizations of parametric polymorphism in C++ and ML for the example of the swap function. For example, which one would be more efficient under which circumstances? (5 Points)*

Answer:

```
(a) fun swap (x,y) =  
    let  
        val tmp = !x  
    in  
        x := !y;  
        y := tmp  
    end;
```

- (b) C++ will generate several instantiations of the code, which will result in more specialized and (probably) faster code. ML produces one version of the code, which operates polymorphically on different types. If multiple instantiations (for multiple types) are needed then the C++ template can result in an explosion of code size.

4 Static vs Dynamic Scope

Same as Mitchell's exercise 7.8.

(10 Points)

Answer:

- (a) Static Scoping

line 2: The x from line 1 is used.

line 4, 5: The x from line 3 is used.

The value of the code is: 16

- (b) Dynamic Scoping

line 2, 4, 5: The x from line 3 is used.

The value of the code is: 21

5 Tail Calls

Take a look at the following implementation of the factorial function with use of an accumulator.

```
fun fact n =  
let  
  fun fact_w_acc 0 a = a  
    | fact_w_acc n a = fact_w_acc (n-1) (n*a)  
in  
  fact_w_acc n 1  
end;
```

- (a) Write down the imperative equivalent of this function. Submit your solution using CheckMate (filename `imperative_factorial.ml`). Briefly describe how your imperative code is equivalent to `fact`. (5 Points)
- (b) Illustrate the activation records on the stack at each function invocation of `fact`. **Hand in the drawing on paper on Friday, September 10th.** (5 Points)

Answer:

```
(a) fun fact x =  
  let val n = ref x and a = ref 1 in  
    while !n > 1 do  
      (a := !a * !n; n := !n - 1);  
    !a  
  end;
```

Instead of using recursion to write the factorial function, the imperative method uses a loop with int refs. It basically loops `n` times, multiplies the current value of `i` (initially set to the value of `n`) times the product, and decrements `i`. This will produce the same results as the recursive factorial function. Also since the recursive function can be optimized due to tail-recursion (see (b)), there will be only one activation record on the stack. This is similar in layout to having to ref

cells.

(b)

(1)	fact 3	access link	(0)	→ Closure for fact_w_acc
		n	3	
		fact_w_acc		
(2)	fact_w_acc 3 1	access link	(1)	
		n	3	
		a	1	

(1)	fact 3	access link	(0)	→
		n	3	
		fact_w_acc		
(2)	fact_w_acc 2 3	access link	(1)	
		n	2	
		a	3	

(1)	fact 3	access link	(0)	→
		n	3	
		fact_w_acc		
(2)	fact_w_acc 1 6	access link	(1)	
		n	1	
		a	6	

(1)	fact 3	access link	(0)	→
		n	3	
		fact_w_acc		
(2)	fact_w_acc 0 6	access link	(1)	
		n	0	
		a	6	

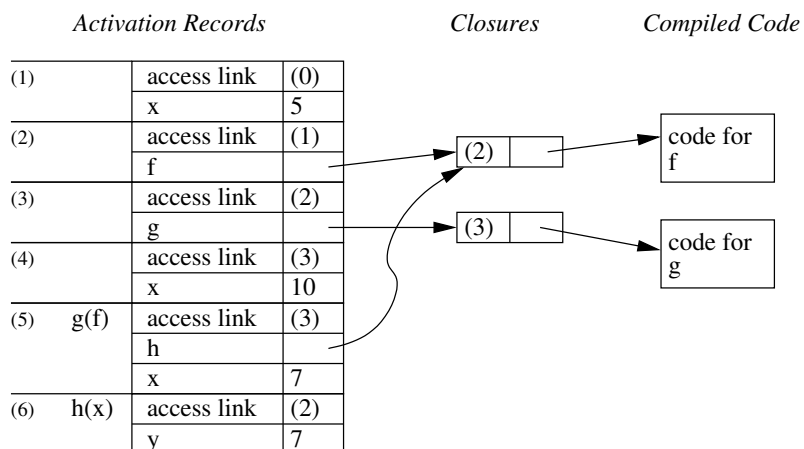
6 Closures

This is the same as 7.12 and 7.13 in Mitchell. **Hand in the drawings (7.12 (a) and 7.13 (b)) on paper on Friday, September 10th.** (20 Points)

Answer:

7.12 (a)

Solutions to Homework 4

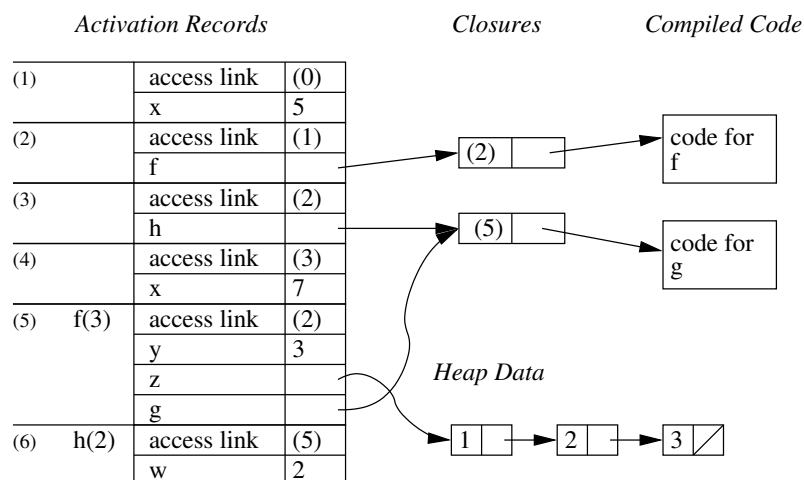


7.12 (b) The value of this expression is 10. ML uses static scoping rules in this expression do determin the value of each of the variables. So in the defintion of f , the x in that expression is going to be 5. The value of x in the function g is going to be 7 according to static scoping. So when $g(f)$ is invoked, the $x = 10$ is essentially ignored.

7.13 (a) x is of type `int`.
 h is of type `int → int`.
 f is of type `int → int → int`.

It should be obvious to see that the type of x is `int`. As for the type of h , we see that it is a function that is taking an `int` and returning an `int`. Hence the type is `int → int`. Finally, h is returned by the function call to f . We also see that function f is taking one `int` as an argument; thus it is a function that takes an `int` argument and returns a function from `int` to `int` giving it the type: `int → (int → int)`. Since the parenthesis are implicit at their current place, we can omit them.

7.13 (b)



- 7.13 (c) Using static scoping, x within the function g is determined to be 5. When $f(3)$ is called, y within function f (same y in function g) is set to 3. h is now a function of the form $((w : \text{int}) + 5 + 3)$. The last line replaces w with 2, hence the final result is 10.
- 7.13 (d) In Mark-and-Sweep garbage collection, we first set all tag bits to 0. Then, start from each location used directly in the program, follow all links and change the tag bit of each cell visited to 1. Finally, place all cells with tags still equal to 0 on the free list.

However, f contains z which is a list with values 1, 2 and 3. Although z is never access by f and h , it is linked to h through f and the Mark-and-Sweep garbage collection will set the tag bit for z to be 1. Hence, as long as h is reachable, mark-and-sweep can never collect z which will never be accessed by the program.

7 Exceptions

```
(a) exception Exn of int;
fun twice(f,x) = f(f x) handle Exn x => x;
fun pred x = if x = 0 then raise Exn x else x - 1;
fun dumb x = raise Exn x;
fun smart x = 1 + (pred x) handle Exn x => 1;
```

What's the result of evaluating each of the following expressions? (10 Points)

- (1) `twice(pred,1);`
- (2) `twice(dumb,1);`
- (3) `twice(smart,1);`

In each case be sure to describe which exception is raised and where, i.e., in the inner/first or outer/last invocation of `twice`?

(b) `exception InstructiveExn of string;`

```

val v = "statically outer var";
let
  fun mid_scope_function () = raise InstructiveExn v
in
  let
    val v = "statically inner var"
  in
    mid_scope_function ()
    handle InstructiveExn x => (x, "statically inner handler")
  end
end handle InstructiveExn x => (x, "statically outer handler");

```

How come that mid_scope_function raises an exception saying ("statically outer var", "statically inner handler")? (5 Points)

Answer:

- (a) (1) `twice(pred, 1)` produces the answer 0. The exception is raised by the outer invocation of `f` within `twice`.
- (2) `twice(dumb, 1)` produces the answer 1. The exception is raised by the first invocation.
- (3) `twice(smart, 1)` produces the answer 1. No exception is raised during the evaluation.
- (b) The code is constructed to illustrate the difference between statically scoped variable lookup and dynamically scoped exception handling: The `InstructiveExn` takes as argument a variable `v` which is statically scoped; and the exception handlers identify their dynamically determined location.

8 Abstract Data Types

Implement the ADT stack in ML for the following interface.

```

type 'a stack
val stack = fn : unit -> 'a stack
val empty = fn : 'a stack -> bool
val top = fn : 'a stack -> 'a
val push = fn : 'a stack * 'a -> 'a stack
val pop = fn : 'a stack -> 'a stack

```

- (a) *Submit your implementation in a file `stack.ml`. (5 Points)*
- (b) *Which design decisions did you have to make? Shortly say why you made these implementation decisions. (5 Points)*

Answer:

(a) `exception StackEmpty;`

```
abstype 'a stack = C of 'a list with
  fun stack('a) = C(nil:'a list)
  fun empty (C(x)) = x = nil
  fun top(C(x)) = if x = nil then raise StackEmpty
                  else hd x

  fun push(C(x), y) = C(y::x)
  fun pop(C(x)) = if x = nil then raise StackEmpty
                  else C(tl x)

end;
```

(b) As an underlying data structure this implementation is using a list. A ML list has quite some similarities with a stack. This first element can be easily examined and removed. Furthermore it only allows to add elements at one end. Hence, it is kind of natural to implement a stack using a list.

The second decision involved the usage of an exception to signal that the stack is empty.

9 Object-Orientation

```
...
Bird b1, b2, b3,b4;
b1 = new Eagle();
b2 = new Sparrow();
b3 = new Penguin();
b4 = new Eagle();

try{
  b1.fly();
  b2.fly();
  b3.fly();
  b4.fly();
}
catch EvolutionReversion {
  print "Action impossible due to evolutionary reversion!\n";
}
```

The above C++/Java pseudo-code excerpt should produce the following output.

```
Eagle flies high.
Sparrow flies fast.
Action impossible due to evolutionary reversion!
```

(a) *To this end, the code assumes certain classes and methods. Write these in Java or C++ and (slightly) adapt the above code to run within your program. Make sure that the produced output is as desired. Submit your program as `birds.java`*

Solutions to Homework 4

or `birds.cpp`, depending on your choice of implementation language. (In the Java case, it should be possible to keep all classes in one file. If not, let us know asap please.) (5 Points)

(b) Shortly describe the class hierarchy and describe where the dynamic lookups happen. (5 Points)

Answer:

(a) **Java based solution**

```
class EvolutionReversion extends Exception {}

public class birds
{
    public static void main(String[] args)
    {
        Bird b1, b2, b3, b4;
        b1 = new Eagle ();
        b2 = new Sparrow ();
        b3 = new Penguin ();
        b4 = new Eagle ();

        try
        {
            b1.fly ();
            b2.fly ();
            b3.fly ();
            b4.fly ();
        }
        catch (EvolutionReversion e)
        {
            System.out.println
                ("Action impossible due to evolutionary reversion!");
        }
    }
}

class Bird
{
    public Bird () {}

    public abstract void fly () throws EvolutionReversion;
}

class Eagle extends Bird
{
    public Eagle () {}
}
```

Solutions to Homework 4

```
        public void fly () throws EvolutionReversion
        {
            System.out.println ("Eagle flies high.");
        }
    }

class Sparrow extends Bird
{
    public Sparrow () {}

    public void fly () throws EvolutionReversion
    {
        System.out.println ("Sparrow flies fast.");
    }
}

class Penguin extends Bird
{
    public Penguin () {}

    public void fly () throws EvolutionReversion
    {
        throw new EvolutionReversion ();
    }
}
```

C++ based solution

```
#include <iostream>

// exception for evolution reversion
class EvolutionReversion {
};

// the parent bird class
class Bird {
public:
    // pure virtual function for dynamic binding
    virtual void fly() = 0;
};

// the inherited classes
class Eagle : public Bird {
public:
    void fly() { std::cout << "Eagle flies high.\n"; }
};

class Sparrow : public Bird {
public:
    void fly() { std::cout << "Sparrow flies fast.\n"; }
};
```

Solutions to Homework 4

```
};

class Penguin : public Bird {
public:
    // penguin cannot fly!!
    void fly() { throw EvolutionReversion(); }
};

int main() {
    // pointers to create new "birds"
    Bird *b1, *b2, *b3, *b4;

    // creating new instances of "birds"
    b1 = new Eagle();
    b2 = new Sparrow();
    b3 = new Penguin();
    b4 = new Eagle();

    // tell the "birds" to fly
    try {
        b1->fly();
        b2->fly();
        b3->fly();
        b4->fly();
    } catch( EvolutionReversion ) {
        // exception detected
        std::cout <<
            "Action impossible due to evolutionary reversion!\n";
    }

    return 0;
}
```

- (b) The class Bird is the superclass of the subclasses Eagle, Sparrow and Penguin. All of the classes have the method fly. The definition of the method fly of the subclasses overrides the method fly of the superclass.

The dynamic lookups happen when fly methods are called within the try block. Since b1, b2 and b3 are declared as different subclasses of the superclass Bird, different fly methods with different definitions are called according to the definition of the fly method within the subclass. The dynamic lookup will find the corresponding fly method.