Design and Implementation of Programming Languages

Prof. Mehran S. Fallah

Assignment #4

Algol / Algol 60 / ML / Reference Cells / L-Values & R-Values / Tagged Union / ML Data Types / ML Patterns / Lists, Records, Tuples, etc.

Done with **love** by

Ali Gholami

with Stu. #

9531504

Department of Computer Engineering and Information Technology

Amirkabir University of Technology

December 2017

The following are problems from the book, **Concepts in Programming Languages, John Mitchell**.

5.1 **Algol 60 Procedure Types**

In Algol 60, the type of each formal parameter of a procedure must be given. However, $proc$ is considered a type(the type of procedures). This is much simpler than the ML types of function arguments. However, this is really a type loophole; because calls to procedure parameters are not fully type checked, Algol 60 programs may produce run-time type errors.

Write a procedure declaration for Q that causes the following program fragment to produce a run-time type error:

$proc\ P\ (proc\ Q)$

$\quad begin\ Q(true)\ \ end;$

$P(Q);$

where true is a Boolean value. Explain why the procedure is statically type correct, but produces a run-time type error.

Answer.

According to *this* report by the leaders of Algol 60, the procedure declarations are written in the following way in Algol 60.

$< procedure\ declaration > ::= \textbf{procedure}\ < procedure\ heading > < procedure\ body > | < type > \textbf{procedure} < procedure\ heading > < procedure\ body >$

According to the above declaration rule for procedures, we can have the following program. This is a procedure with $proc$ type for sure, but the resulting value is not a Boolean, which will cause the program to produce a run-time type error.

$\textbf{procedure}\ Transpose\ (a)\ Order: (n); \textbf{value}\ n;$

$\textbf{array}\ a; \textbf{integer}\ n;$

$\textbf{begin real}\ w; \textbf{integer}\ i, k;$

$\textbf{for}\ i := 1\ \textbf{step}\ 1\ \textbf{until}\ n\ \textbf{do}$

$\quad \textbf{for}\ k := 1 + i\ \textbf{step}\ 1\ \textbf{until}\ n\ \textbf{do}$

$\quad \textbf{begin}\ w := a[i, k];$

$\quad\quad a[i, k] := a[k, i];$

$\quad\quad a[k, i] := w;$

$\quad \textbf{end}$

$\textbf{end}\ Transpose$

While trying to pass the above procedure as an argument to the $proc\ P$, being a procedure type, the static type checker will assume it as a true condition and the type of the returning value on the line $begin\ Q(true)\ end;$ won't be evaluated. So, this causes the run-time error. Summarizing the answer, the procedure p accepts a **procedure** as its argument, but the **type** of that **internal procedure** is not declared anywhere.

5.2 Algol 60 Pass-By-Name

The following Algol 60 code declares a procedure P with one pass-by-name integer parameter. Explain how the procedure call $P(A[i])$ changes the values of I and A by substituting the actual parameters for the formal parameters, according to the Algol 60 copy rule. What integer values are printed by $tprogram$? By using pass-by-name parameter passing?

The line integer x does not declare local variables – this is just Algol 60 syntax declaring the type of the procedure parameter.

```
begin
    integer i;
    integer array A[1:2];

    procedure P(x);
        integer x;
        begin
            i := x;
            x := i
        end

        i := 1;
        A[1] := 2;
        A[2] := 3;
        P (A[i]);
        print (i, A[1], A[2])
end
```

Answer.

According to Prof. Fallah sayings, pass-by-name will replace the actual parameters names with the formal ones. So, the resulting criteria will be a procedure which is called with $A[i]$ and x will be equal to $A[i]$. The last modification of i shows its value as 1. So, the first line in the procedure body will assign 2 to the contents of the i and the next line will replace the previous $A[2]$ contents, which was 3, with the contents of i, which is 2.

The line $print(i, A[1], A[2])$ will have "2 2 2" as a result.

5.3 Nonlinear Pattern Matching

ML patterns cannot contain repeated variables. This exercise explores this language design decision. A declaration with a single pattern is equivalent to a sequence of declarations using destructors. For example,

```
val p = (5, 2);
val (x,y) = p;
```

Is equivalent to

```
val p = (5, 2);
val x = #1(p);
val y = #2(p);
```

Where $\#1(p)$ is the ML expression for the first component of pair p and $\#2$ similarly returns the second component of a pair. The operations $\#1$ and $\#2$ are called $destructors$ for pairs.

A function declaration with more than one pattern is equivalent to a function declaration that uses standard $if - then - else$ and $destructors$. For example,

```
fun f nil = 0
| f(x::y) = x;
```

is equivalent to

```
fun f(z) = if z=nil then 0 else hd(z);
```

where hd is the ML function that returns the first element of a list.

(a) Write a function declaration that does not use pattern matching and that is equivalent to

```
fun f(x, 0) = x
|   f(0, y) = y
|   f(x, y) = x + y;
```

ML pattern matching is applied in order, so that when thus function is applied to an argument (a, b), the first clause is used if $b = 0$, the second clause if $b \neq 0$ and $a = 0$ and the third clause if $b \neq 0$ and $a \neq 0$.

Answer (a).

```
fun f(x, y) = if y = 0 then x else
                  if x = 0 then y else x + y;
```

(b) Does the method you used in part (a), combining $destructors$ and $if - then - else$, work for this function?

```
fun eq(x, x) = true
|   eq(x, y) = false;
```

Answer (b).

Using the ML shell, we can verify that the above patterned function won't work. We can understand that patterns with the same name as arguments of the functions cannot be used in ML for some good reasons. The first reason could be the *type errors* or the *type variety* of the arguments being passed. *functions* cannot be always verified through their *type* and they might cause *runtime type error*. For sure, there was a possibility of checking functions *return type* but they could have different *bodies* & *functionalities*. So, statically checking of the functions is not a good idea.

(c) How would you translate ML functions that contain patterns with repeated variables into functions without patterns? Give a brief explanation of a general method and show the result for the function *eq* for in part (b).

Answer (c).

So, the problem described in the previous section can be solved using variables with different names in the function declaration. But, we have to make sure that these two different variable are equal or not. Here, equality means that the two variables (e.g. x and y in $fun\ eq(x,y)$) have the same evaluation result. So, after evaluating each of them, probably, our program will have some states, and (x,y), in order to match, must represent the same state. Using the *destructors* in the question information, we can extract and evaluate the arguments of the $eq(x,y)$ and check if their evaluation results are the same like so.

```
fun eq(x, y) = if #1(x,y)=#2(x,y) then true else false;
```

(d) Why do you think the designers of ML prohibited repeated variables in patterns?

Answer (d).

I guess the enough talk held on the sub-section *b*. The main issue is that static type checking is not enough. Also, dynamic type checking would need more and more complex considerations for the pattern matching. The option of removing this feature rather than handling it (I'm not sure if it is not handled in newer versions) was a logical decision.

5.4 ML Map for Trees

(a) The binary tree data type

$$datatype\ 'a\ tree = LEAF\ of\ 'a\ |$$
$$NODE\ of\ 'a\ tree * 'a\ tree;$$

Describes a binary tree for any type, but does not include the empty tree. Write a function $maptree$ that takes a function as an argument and returns a function that maps trees to trees by mapping the values at the leaves to new values, using the function passed in as a parameter. In more detail, if f is a function that can be applied to the leaves of tree t and t is the tree on the left, then $maptree\ f\ t$ should result in the tree on the right. For example, if f is the function $fun\ f(x) = x + 1$ then

$$maptree\ f\ (NODE(NODE(LEAF\ 1, LEAF\ 2), LEAF\ 3))$$

Should evaluate to $NODE(NODE(LEAF\ 2, LEAF\ 3), LEAF\ 4)$. Explain your definition in one or two sentences.

Answer (a).

The function $maptree$ will be

```
fun maptree(node(x, y), z) = node(maptree(x, z), maptree(y, z))
| maptree(leaf(x), y) = leaf(y(x))
```

(b) What is the type ML gives to your function? Why is it not the expected type $('a \to' a) \to\ 'a\ tree \to 'a\ tree$?

Answer (b).

The resulting type for the above function declaration will be as following.

$$\left('a\ tree * ('a \to\ 'b)\right) \to ('b\ tree)$$

5.6 **Currying**

This problem asks you to show that the ML types $'a \to ('b \to\ 'c)\ and\ ('a \to\ 'b) \to 'c$ are essentially equivalent.

(a) Define higher-order ML functions

$$Curry: \left(('a *\ 'b) \to\ 'c\right) \to ('a \to ('b \to\ 'c))$$

And

$$UnCurry: \left('a \to ('b \to\ 'c)\right) \to (('a *\ 'b) \to\ 'c)$$

Answer (a).

$$-val\ curry = fn\ f => fn\ a => fn\ b => f(a, b);$$

$$-val\ UnCurry = fn\ f => fn(a, b) => f\ a\ b;$$

(b) For all functions $f: ('a * 'b) \rightarrow' c \; and \; g:' a \rightarrow ('b \rightarrow 'c)$, the following two equalities should hold(if you wrote the right functions):

$$UnCurry\big(Curry(f)\big) = f$$
$$Curry\big(UnCurry(g)\big) = g$$

Explain why each is true for the functions you have written. Your answer can be three or four sentence long. Try to give the main idea in a clear, succinct way. Be sure to consider termination behavior as well.

Answer (b).

The following will be the result of applying $Uncurry$ and $Curry$ together on a function f, which will result in f for sure.

$$UnCurry(Curry('a * 'b) \rightarrow 'c)$$
$$UnCurry('a \rightarrow 'b \rightarrow 'c)$$
$$'a * 'b \rightarrow c$$

The following is the result of applying $Curry$ and $UnCurry$ on a function g, which will also result in g for sure.

$$Curry(UnCurry \; 'a \rightarrow 'b \rightarrow 'c)$$
$$Curry('a * 'b \rightarrow 'c)$$
$$'a \rightarrow 'b \rightarrow c$$