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)
   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. (You may assume that adding a Boolean to an integer is a run-time type error.)

```
proc Q (integer x)
begin
   x + true
end;
```

Static Type Correctness: From a static typing perspective, the code seems correct. The procedure P is declared to accept a procedure Q as its parameter, and when P is invoked with Q as an argument, it satisfies the type requirement.

Run-time Type Error: However, the issue arises during execution. The procedure Q expects an integer parameter (x), but it is called with a Boolean value (true). This causes a run-time type error because the code attempts to perform an operation (addition) between an integer (x) and a Boolean (true), which is not supported in Algol 60.

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
```

Following the argument passing to the procedure, the sequence executes as follows: `i` is assigned the value of `A[i]`, then `A[i]` is assigned the value of `i`. This yields the output `2, 2, 2`.

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.

Questions:

(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 this function is applied to an argument (a, b), the first clause is used if b = 0, the second clause if b≠0 and a=0, and the third clause if b≠0 and a≠0.

fun f(x: int*int) =

If #2x = 0 then #1x else if #1x = 0 then #2x else #1x + #2x

(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;
```

Without specifying parameter types explicitly, achieving the desired functionality for the `eq` function using the destructors and if-then-else method is not possible.

(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 in part (b).

Patterns with repeated variables in ML can be translated into functions without patterns by using multiple parameters. For instance, the `eq` function can be transformed into:

`eq_v2(x, y) = if x = y then true else false`.

(d) Why do you think the designers of ML prohibited repeated variables in patterns? (*Hint:* If f, g : int $\longrightarrow$ int, then the expression f = g is not type-correct ML as the test for equality is not defined on function types.)

ML prohibits repeated variables in patterns to maintain clarity and type-safety. Comparing functions for equality, like `f = g` where `f` and `g` are functions, isn't defined in ML due to the complexity of determining function equality. This prohibition avoids ambiguities and undefined behavior in pattern matching and type-checking.

5.6 Currying

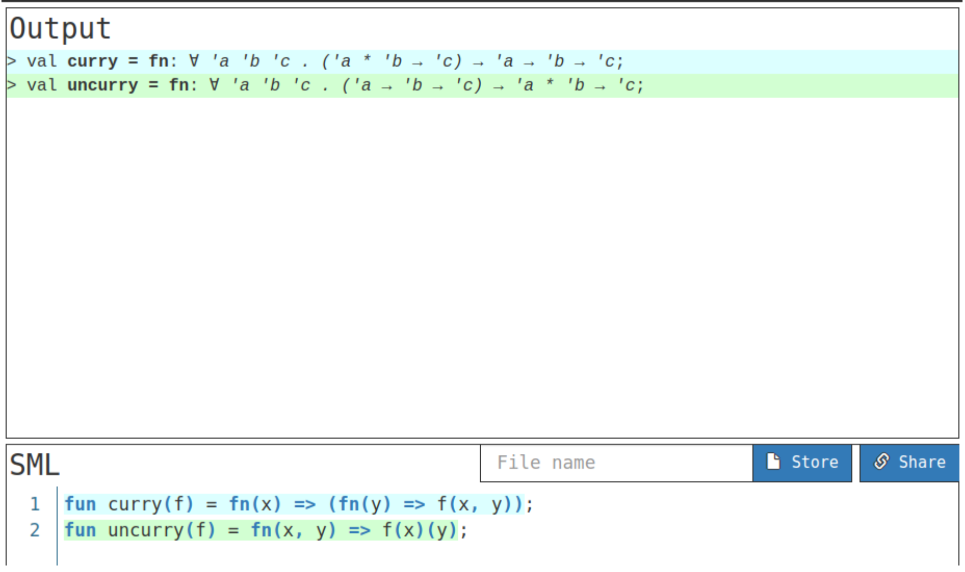
This problem asks you to show that the ML types 'a → ('b → 'c) and ('a ∗ 'b) → 'c are essentially equivalent.

(a) Define higher-order ML functions

Curry: (('a ∗ 'b) → 'c)→ ('a → ('b→ 'c))

and

UnCurry: ('a → ('b → 'c))→ (('a ∗ 'b) → 'c)



Output

```
> val curry = fn: ∀ 'a 'b 'c . ('a * 'b → 'c) → 'a → 'b → 'c;
> val uncurry = fn: ∀ 'a 'b 'c . ('a → 'b → 'c) → 'a * 'b → 'c;
```

SML

```
fun curry(f) = fn(x) => (fn(y) => f(x, y));
fun uncurry(f) = fn(x, y) => f(x)(y);
```

(b) For all functions f : ('a ∗ 'b) → 'c and g : 'a → ('b → 'c), the following two equalities should hold (if you wrote the right functions):

UnCurry(Curry(f)) = f
Curry(UnCurry(g)) = g

Explain why each is true for the functions you have written. Your answer can be three or four sentences long. Try to give the main idea in a clear, succinct way. (We are more interested in insight than in number of words.) Be sure to consider termination behavior as well.

To demonstrate the equivalency between the types 'a → ('b → 'c) and ('a ∗ 'b) → 'c, the functions Curry and UnCurry are defined as follows:

Curry : (('a ∗ 'b) → 'c) → ('a → ('b → 'c))

UnCurry : ('a → ('b → 'c)) → (('a ∗ 'b) → 'c)

Now, for any functions f : ('a ∗ 'b) → 'c and g : 'a → ('b → 'c) , the following equalities hold:

UnCurry(Curry(f)) = f

Curry(UnCurry(g)) = g

The first equality states that if you curry and then uncurry a function, you obtain the original function. This is true because Curry takes a function of type (('a ∗ 'b) → 'c) and converts it into a curried form ('a → ('b → 'c)), and UnCurry does the reverse, transforming the curried function back into its original form.

The second equality asserts that if you uncurry and then curry a function, you also obtain the original function. This holds true because UnCurry takes a curried function ('a → ('b → 'c)) and converts it into a function of type (('a ∗ 'b) → 'c), and Curry does the reverse, transforming the uncurried function back into its original form.

The termination behavior is guaranteed to be preserved in these transformations, as Curry and UnCurry only manipulate the structure of the functions without affecting their underlying behavior. Therefore, these equalities hold, demonstrating the essential equivalence between the two types.

The second equality asserts that if you uncurry and then curry a function, you also obtain the original function. This holds true because UnCurry takes a curried function ('a → ('b → 'c)) and converts it into a function of type (('a ∗ 'b) → 'c), and Curry does the reverse, transforming the uncurried function back into its original form.

The termination behavior is guaranteed to be preserved in these transformations, as Curry and UnCurry only manipulate the structure of the functions without affecting their underlying behavior. Therefore, these equalities hold, demonstrating the essential equivalence between the two types.

5.7 Disjoint Unions

A *union type* is a type that allows the values from two different types to be combined in a single type. For example, an expression of type union(A, B) might have a value of type A or a value of type B. The languages C and ML both have forms of union types.

(a) Here is a C program fragment written with a union type:

```
...
union IntString {
   int i;
   char *s;
} x;
int y;
if ( ... ) x.i = 3 else x.s = “here, fido”;
...
y = (x.i) + 5;
...
```

A C compiler will consider this program to be well typed. Despite the fact that the program type checks, the addition may not work as intended. Why not? Will the run-time system catch the problem?

The program uses a union type in C, where the union IntString can hold either an integer int i or a string char *s. The code sets the value of the union x based on a condition, either initializing the integer field x.i = 3 or the string field x.s = "here, fido".

The issue arises in the line:

y = (x.i) + 5;

This line assumes that the current active field of the union is i, and it tries to add 5 to it. However, if the condition in the earlier part of the code led to setting the string field s, then accessing x.i would result in undefined behavior because it interprets the bits of the string pointer as an integer.

The C compiler allows this code to compile because unions in C share the same memory space for all their members. The type-checking at compile-time is based on the assumption that you, as the programmer, are responsible for ensuring that you access the correct field based on the program logic.

The run-time system may not catch this problem because C does not perform runtime checks on the active field of a union. It is the programmer's responsibility to track the active field and access it appropriately. If you access the wrong field, it may result in unexpected behavior, including crashes, data corruption, or other issues, but these won't necessarily be caught by the run-time system. Therefore, caution is required when using unions to avoid such pitfalls.

(b) In ML, a union type union(A,B) would be written in the form datatype UnionAB = tag_a of A | tag_b of B and the preceding if statement could be written as

```
datatype IntString = tag_int of int | tag_str of string;
...
val x = if ... then tag_int(3) else tag_str(“here, fido”);
...
let val tag_int (m) = x in m + 5 end;
```

Can the same bug occur in this program? Will the run-time system catch the problem? The use of tags enables the compiler to give a useful warning message to the programmer, thereby helping the programmer to avoid the bug, even before running the program. What message is given and how does it help?

In this case, the bug you described in the C program is less likely to occur due to the use of tags. The tags explicitly indicate which constructor is used to create the value, making it clear which type is currently active.

ML is a strongly typed language with pattern matching, so the type system helps catch such errors at compile-time. If there's a mismatch in the pattern matching, the ML compiler will generate an error, and the program will not compile. Therefore, such issues are caught before runtime.

The error message from the compiler would likely indicate a pattern-matching failure, specifying that you are missing a case in your pattern match or trying to access a field that doesn't exist for the given tag. This helps the programmer identify and rectify the issue during development, preventing runtime errors. The message is typically informative and directs the programmer to the specific location in the code where the problem occurs.