# Notes on Unions

### Typical Union Example, Comparing C and FMSL

The way FMSL unions work is very much like unions in C. The key difference is that an FMSL union has a built-in tag that C unions don't have. The interpreter maintains the value of the tag internally, to keep track of which union component is currently in use.

An important syntax change that is not properly updated in the reference manual is the tag-interrogation operator. Previously it was just '?', but it has been updated to '?.'. The upcoming examples show how to use it.

What follows is a pretty typical use of union types in C and FMSL. It's the same union type in both cases, with very similar processing. The main difference is the automatic tag maintenance done in FMSL. Both examples are compilable, in the files `union-example.c` and `union-example.fmsl`, in the directory `fmsl/documentation/contrib/pcorwin`.

**C Union, with Programmer-Maintained Tags:**

```
#define bool unsigned char
#define true 1

typedef enum {INT, BOOL, STRING} ValTag;

typedef struct {
    ValTag tag;
    union {
        int intVal;
        bool boolVal;
        char* stringVal;
    } val;
} IntOrBoolOrString;

IntOrBoolOrString ibs;
int i;
bool b;
char* s;

int main(int argc, char** argv) {

    /*
     * Use ibs as an int.
     */
    ibs.val.intVal = 10;        // Assign the intVal field
    ibs.tag = INT;              // Keep track that ibs is currently an int
    i = ibs.val.intVal;         // Assign 10 to i, as expected

    /*
     * Erroneously use ibs as a string, with int as the current component.
     */
    s = ibs.val.stringVal;      // Assigns effective garbage to s; it's the
                                // programmer's responsibility to keep track of
                                // which union field is currently active

    /*
     * Use ibs as a boolean, again with programmer-maintained change of tag.
     */
    ibs.val.boolVal = true;
    ibs.tag = BOOL;
```

```
    b = ibs.val.boolVal;

    /*
     * Use ibs as a string, again with programmer-maintained change of tag.
     */
    ibs.val.stringVal = "xyz";
    ibs.tag = STRING;
    s = ibs.val.stringVal;

    /*
     * Here is the normal way to safely use a union variable.  It involves
     * checking the current value of the tag, and accessing the field that the
     * tag-check says is currently active.  This is the place in C where the
     * programmer-maintained tag value used.  There are a number of ways to
     * implement it.  The use of an enum like ValTag is pretty typical.
     */
    if (ibs.tag == INT) {
        i = ibs.val.intVal;     // Safely fetch the int value
        // do whatever with i
    }
    if (ibs.tag == BOOL) {
        b = ibs.val.boolVal;    // Safely fetch the bool value
        // do whatever with b
    }
    if (ibs.tag == STRING) {
        s = ibs.val.stringVal;  // Safely fetch the string value
        // do whatever with s
    }

    /*
     * Try to assign an integer to the whole union variable.
     */
    // ibs = 10;                // Produces a compiler type error.

}
```

**FMSL Union, with Interpreter-Maintained Tags:**

```
obj IntOrBoolOrString = intVal:integer or boolVal:boolean or stringVal:string;

var ibs:IntOrBoolOrString;
var i:integer;
var b:boolean;
var s:string;

op main() = (
    (*
     * Use ibs as an integer; here the interpreter keeps track of the fact that
     * the intVal component is currently active.
     *)
    set ibs.intVal = 10;        -- Treat ibs as an int; in contrast to C, the
                                -- interpreter internally maintains the tag,
                                -- which is set to indicate that the intVal
                                -- component is currently active
    set i = ibs.intVal;         -- Assign 10 to i, as expected

    (*
     * Erroneously use ibs as a string, with int as the current component.
     *)
    set s = ibs.stringVal;      -- Assigns nil to s, since the interpreter
                                -- knows that ibs is currently an integer
```

```
    (*
     * Use ibs as a boolean, again with interpreter-maintained change of tag.
     *)
    set ibs.boolVal = true;
    set b = ibs.boolVal;

    (*
     * Use ibs as a string, again with interpreter-maintained change of tag.
     *)
    set ibs.stringVal = "xyz";
    set s = ibs.stringVal;

    (*
     * Here is the normal way to safely use a union variable.  It involves
     * checking the current value of the tag, and accessing the component that
     * the tag-check says is currently active.  This is the place in FMSL where
     * the '?.'  operator gets used.  It returns true for the currently used
     * field, false for all of the other fields.
     *)
    if ibs?.intVal then (
        set i = ibs.intVal;     -- Safely fetch the integer value
        -- do whatever with i;
    );
    if ibs?.boolVal then (
        set b = ibs.boolVal;    -- Safely fetch the boolean value
        -- do whatever with b
    );
    if ibs?.stringVal then (
        set s = ibs.stringVal;  -- Safely fetch the string value
        -- do whatever with s;
    );

    (*
     * Assign an integer to the whole union variable.  This is called
     * "auto-injection", and is not supported by C.  DO NOT IMPLEMENT IT.
     *)
    set ibs = 10;
);
```

As the comments explain, the FMSL interpreter keeps track of the type of the the currently in-use component of a union. In this context, "in-use" means the component most recently accessed with the '.' operator. So, in a ValueStruct for a union type, there is (at least) one Value* for the current component, and a tag that indicates which component is current.

You can implement this in the interpreter in a number of ways. A quick and dirty way is to use tuple values almost as-is for union values. The only change is to add a tag field, that the interpreter can use to keep track of the currently active field. In this solution, the ValueStruct for a union value is represented as the full tuple, plus the current tag. Whenever the '.' operator is used on a union value, the tag field is set to the component named on the right of the '.'. Otherwise, the implementation of '.' is the same for unions as it is for tuples. And it's the ValueStruct itself that keeps track of the current component, independent of what variable the value is bound to.

A more space-efficient representation would have only one Value* for the current value, instead of the array of Value* used for tuples. Unless you find this representation conceptually easier to implement, it's not worth implementing just for the storage saving.

Looking at the Money example in your email

```
obj Money = coins:integer or dollars:integer;
```

it's legal in FMSL, just like it would be in C

```
typedef enum {COINS, DOLLARS} MoneyTag;
typedef struct {
    MoneyTag tag;
    union {
        int coins;
        int dollars;
    } val;
} Money;
```

Having a union with two components of the same type probably isn't all that useful. That said, the way you'd access the coins and dollars fields would be just like in the examples above. I.e., m?.coins returns true if the coins field is active, and m.coins accesses this field. If coins is in fact the active field, then m.coins returns whatever value is stored there. If m.coins is not current, then m.coins returns nil.

### Auto-Injection

The last line of in the FMSL union example illustrates what's called "auto-injection". "Injection" is the formal term frequenting used for binding one of the values of a union type to a union value itself. The current FMSL type checker supports this, but it would be a pain to implement. So, I'd like you to skip it, so you can get to the testing stuff more expeditiously.

### FMSL Unions as Enums

There are a lot of FMSL examples that use union types to do what enums do in C. Here is another side-by-side comparison of C and FMSL, showing a typical enumerated type. The examples are in the files `enum-example.c` and `enum-example.fmsl`.

**C Enum for Days of the Week**

```
typedef enum {
    Sun,
    Mon,
    Tue,
    Wed,
    Thu,
    Fri,
    Sat,
} DaysOfWeek;

DaysOfWeek d;

int main(int argc, char** argv) {

    if (d == Sun) {
       // Do Sunday processing
    }
    if (d == Mon) {
       // Do Monday processing
    }
    // etc. for other days.

    /* Here's the normal way to assign an enum value to a variable. */
    d = Wed;

}
```

**FMSL Enum for Days of the Week**

```
obj DaysOfWeek = sun:Sun or mon:Mon or tue:Tue or wed:Wed or thu:Thu or
                 fri:Fri or sat:Sat;
obj Sun;
```

```
obj Mon;
obj Tue;
obj Wed;
obj Thu;
obj Fri;
obj Sat;

var d:DaysOfWeek;

op main() = (

    if d?.sun then (
       -- Do Sunday processing
    );
    if d?.mon then (
       -- Do Monday processing
    );
    -- etc. for other days.

    (* Here's how to assign an emum value to a variable, given this style of
     * enum in FMSL.  Getting this to work also involves auto-injection. *)
    set d = 'Wed';

);
```

As the comment for the last line again explains, this example requires auto-injection to work properly. If necessary, I can get auto-inject to work reasonably quickly.

**Unions of Concrete Values**

A topic we've not discussed directly is tuples and unions with components that are concrete values. Here's another version of the DaysOfWeek enumeration that uses this feature. (It's in the file `enum-string-example.fmsl`.)

```
obj DaysOfWeek = "Sun" or "Mon" or "Tue" or "Wed" or "Thu" or "Fri" or "Sat";

var d:DaysOfWeek;

op main() = (

    if d = "Sun" then (
        -- Do Sunday processing
    );
    if d = "Mon" then (
        -- Do Monday processing
    );
    -- etc. for other days.

    (* Here's how to assign an emum value to a variable, given this style of
     * enum in FMSL. *)
    set d = "Wed";

);
```

As with the auto-inject feature, you DON'T need to implement constant-valued tuples and enums; the testing work can be showcased fine without this functionality. The good news is that these features should pretty well work with the implementation of tuples and unions you will have completed already, i.e., without having focused specifically on constant-valued tuple and union components.