Notes on Updated Memory Model
2 February 2009

The memory model has been fundamentally changed, from the compiler-like Modula-2 model, to a uniform model based on arrays of Value pointers.

In the old model, memory was laid out as raw data values, of differing sizes, aligned on byte boundaries. For example, consider a memory segment of integer, boolean, and real. The size of this segment was sizeof(integer) + sizeof(bool) + sizeof(double), which is typically 13 bytes. Identifiers bound to these respective values were given byte offsets of 0 (for the int), 4 (for the bool), and 5 (for the real). To access a identifier's value:

a. lookup the identifier in the current symbol table

b. get it's memory offset from the symbol table entry

c. memcpy the data starting at the offset, copying TypeSize(value) bytes, where TypeSize = 4, 1, or 8 in this example.

E.g., the real-valued data in this example are 8 bytes, starting at offset 5.

In this context, an "identifier" is some name designating a storage location. In the interpreter, there are three kinds of storage-designating identifiers --

a. operation parameters and local variables

b. tuple field names

c. global variables.

Details of the memory layout for these three kinds of idents are discussed as we go.

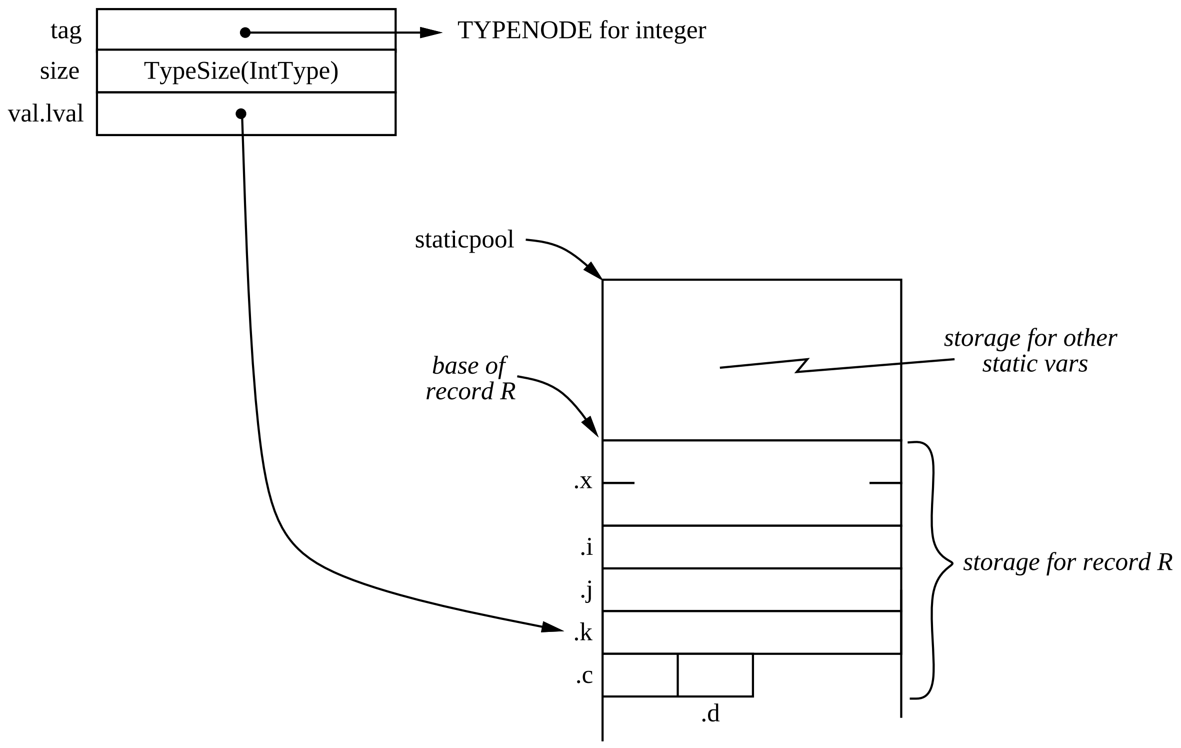
In the new memory model, all values are uniformly represented as pointers to a Value struct, of sizeof(Value*), which typically is 4 bytes. In the preceding memory example, the new segment size is 3 * sizeof(Value*). Memory offsets are monotonically-increasing by increments of sizeof(Value*). I.e., identifiers are bound to offsets 0 (for the int), 1 (for the bool), and 2 (for the real).

Also in the new memory model, the first two access steps are the same as in the old model, i.e., lookup an ident and get it's offset. But the third access step does not require a memcpy to get the data. Rather, the data are accessed by following a Value pointer, into its val component. E.g, the real-valued data are accessed in the val.RealValue component, of the Value pointed to in the variable at memory offset 2.

To summarize, what the new memory model looks like is a C array of Value struct pointers. E.g., instead of a raw 4-byte integer in memory, there's a pointer to a Value that holds the integer. This is obviously a lot less efficient, both in terms of space and access time. But time and space of this nature really don't matter much, if at all, in the kind of interpreter we're building. Plus the uniform structure makes things much easier to deal with in FMSL compared to Modula. Among the issues that the uniform memory simplifies are the declare-after-use semantics of FMSL, recursive types, and value binding.

Here are a couple pictures that illustrate the old and new memory structures The first pictures is taken from 451 Lecture Notes Week 3. It illustrates the memory layout for the following chunk of Modula-2 code:

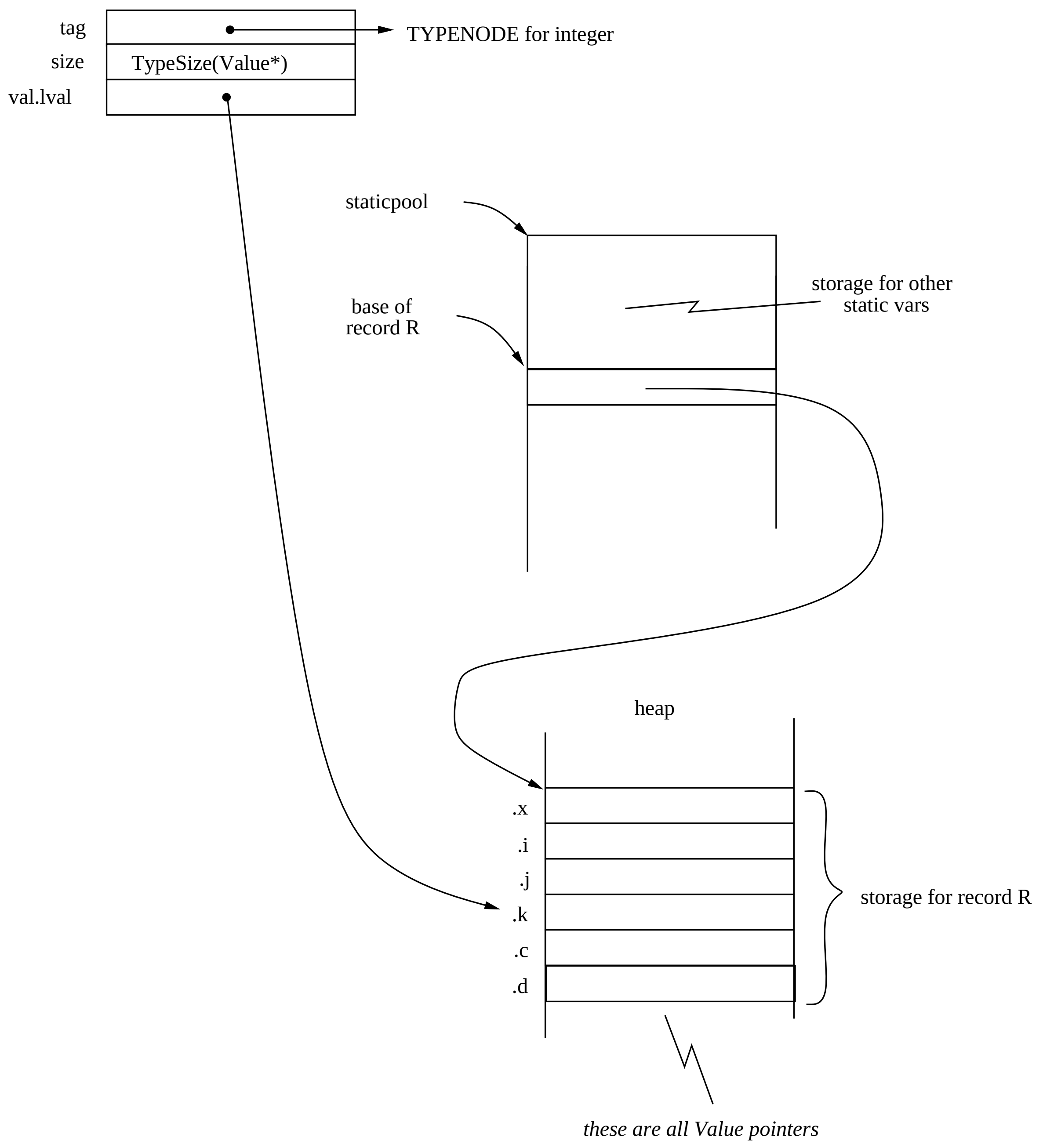
```
module
    ...
    var R:
        record
            x: real;          /* 8 byte real */
            i,j,k: integer;   /* 4 byte ints */
            c,d: char;        /* 1 byte chars */
        end;
begin
    R.k := 10;
end
```



The next picture is for the revised memory layout, for the equivalent FMSL chunk of code:

```
obj rec =
    x:real and
    i:integer and j:integer and k:integer and
    c:string and d:string;
var R:rec;

> R.k := 10;
```



The last line of FMSL code in fact legal, but it’s not part of the functional core of the language, and not really the focus of what we’re doing. But since we’re working from the old Modula-2 interp, the code for designators will allow it.

The important point for what you’re doing is that the existing `doDesignator` and related functions should work with one modification -- change the memcpy to just a regular C assignment statement. That is, instead of memcopying, a segment of bytes from memory into a Value, just assign the value pointer to the designated memory location.

Three Storage Areas

Corresponding to the three different kinds of designating idents, there are three storage areas that are specifically affected by the memory model:

a. operation activation records
b. tuple storage
c. the static pool of globally-declared variables

Each of these will now be represented simply as a C array of Value*. The sizes of these areas are, respectively:

a. the number of input parameters, output parameters, and local variables for an operation
b. the number of top-level components of a tuple
c. the total number of globally-declared variables

Some details regarding each of these areas follow.

Activation Records

When the type checker checks an operation/function declaration, it creates a symbol table for the operation's scope. It makes an entry in the symtab for each input parameter, each output parameter, and each local 'let' declaration. A 'let' expression declares a local variable, as well as assigning it a value.

The interpretation of operation invocation can go essentially as it is for Modula-2, given that the parameter and local var offsets are pre-computed. With the new memory model, you don't have to distinguish between the different kinds of parameters that were in Modula, i.e., call-by-var and open array. So, you can get rid of all the code that deals with those parameter styles. Everything is strictly call-by-value, and each parameter takes a uniform amount of storage, namely the storage for a Value* that points to the actual parameter value.

Tuples

A tuple is a block of memory much like an activation record. The type checker creates a symtab for each tuple, and stores it in the tuple type tree. Specifically, for a tuple type t, the symtab and its size are stored in

```
t->components.type.kind.record.fieldstab
```

and

```
t->components.type.kind.record.numfields
```

Here "record" is the old name for "tuple", left over from Modula-2. I tried changing the name, but it broke a bunch of stuff, so I figure we can just live with the slight misnomer.

In the new uniform memory layout, each tuple component takes exactly the same amount of storage, that is, the size of a Value*. Another significant difference from the old Modula memory scheme is the memory for nested tuples. Specifically in the new model, a nested tuple does not store the sub-components inline. For example, in a Modula-2-style memory scheme, the following nested tuple would have had inline storage for five integers

```
object NT = c1:integer and c2:integer and t:(
                i1:integer and i2:integer ant i3:integer);
```

In the new scheme, NT has storage for three Value*, which are the top-level components of the type. The third top-level component has a Value* that points in turn to a three-component sub-tuple.

You can think of tuples as arrays of Value*, indexable by a field name for each array element. The field-name-to-array-index correspondence is the offset stored in the tuple's symbol table for each tuple component. E.e., for NT, the c1 component is at offset 0, c2 at offset 1, and t at offset 2. The offset is used directly as an array index to get the Value for each component.

The Global Static Pool

In the Modula-2 interpreter, global vars played a more fundamental role than they do in FMSL. There is a global var declaration in FMSL, but using it crosses the line from a functional/declarative spec to a non-functional/procedural spec. Nevertheless, I think it's easy enough to implement global vars, given the Modula-2 base interp. In particular, the binding of operation parameters is done by assignment. Also, it will be handy to have global vars for testing purposes

Syntactically, a global var declaration looks like this:

```
    var x:type [= value]
```

In the current interp.h, the static pool size is declared as 25000. The type checker has a count of the number of global vars declared in the files it checks, but when the interp runs interactively, new globals can be declared. So, 25K is an initial size estimate, to allow for a generous number of global var decls. There's code in the interp to increase the size of the pool if more space is needed as an interactive session proceeds.

In the old memory model, static storage for all non-pointer vars was totally pre-allocated in static pool. "Allocated" means that the static pool offset ended up being at the physical end of all allocated storage. In the case of Modula arrays and records, all storage was allocated in-line within the static pool. So, to access an array value or record field, you just moved an offset index to an appropriate place in the pool.

In the new memory model, list and tuple storage will only be initially allocated as a single pointer per list or tuple. The actual storage for the list or tuple will be allocated when the list or tuple is constructed with the constructor ops: [...] and {...}.

Tuple Value Construction

The new memory scheme prominently affects the way tuple construction is implemented. Consider the following example, that declares and initializes a variable of type NT:

```
     var nt:NT = {1,2,{10,20,30}};
```

In the interp, there can be a function called doTupleConstructor, that allocates Value structs for a tuple value. In the case of this example, the variable nt will only have storage for one Value* pointer in the static pool. Again, static pool "allocation" means that the variable nt has an offset into the pool, and that offset is at the beginning has a one-word Value* pointer.

The following happens in the doTupleConstructor function:

a. a record (aka, tuple) Value is malloc'd for the outer tuple; this value is an array with three Value* elements
b. the pointer to newly malloc'd tuple array is stored in nt's static pool location
c. an integer Value is created for the first two component values -- 1 and 2
d. these two values are pointed to from the first two Value* pointers in nt's array
e. another 3-element record Value is created for the nested tuple, i.e., a C array with three Value* elements
f. the third Value* of nt points to the new sub-tuple Value
g. finally, an integer Value is created for the three sub-component values -- 10, 20, 30, and they're pointed to from the three elements of the sub-tuple array

Up at the top of the structure, the Value for nt will contain the complete type for the nested tuple, including the symtab the the type checker builds. Given this, I'm hoping that the function RecordRef can work as-is, since it uses the tuple symtab to find the offset of a record field, i.e., tuple component. What will come back from RecordRef an l-value that points to a selected field. (Clerically, RecordRef could be renamed TupleRef, but there are still plenty of places where "record" appears in names. I think we just need to live with the fact that "record" and "tuple" are synonyms.)

L-Values versus R-Values for Designated Storage

Let's talk about this in our next phone meeting.