Tuple Abstract Data Type

Gunnar Gotshalks

2007 December

Table of Contents

1 Introduction

The value of defining tuples as an abstract data type is primarily for systems programmers. We do not go into detailed program implementations.

A tuple is a heterogeneous collection of variables of different types. By combining the variables into a single unit, it becomes possible to abstract information into larger units and to pass a smaller amount of explicit information among abstract data type operations. Note: tuples are also called records.

Arrays and tuples are complementary in that arrays are a homogenous collection of variables all of the same type, while tuples are a heterogenous collection of variables of potentially different types. This complementary aspect also appears in the ease with which elements from the structure can be accessed. For arrays it is simple; for tuples it is more complex. That is why the tuple structure is less often a part of programming languages than arrays.

In the C and C++ languages tuples are represented by the struct statement. In object–oriented languages tuples are represented by a class with attributes only; no routines. The underlying representation of routine parameters and calling arguments are tuples.

2 The tuple object

The entire tuple is a single entity, represented by its name.

3 The tuple fields

Each tuple field is a separate object. Figure 1 is a memory representation of a tuple.

Each field can be of a different type (represented by the colours) and of a different size (represented by the different sizes of the areas.

Figure 1: A prototypical tuple descriptor.

Tuple fields are represented using the following schema.

```
tuple_name.field_name
```

For example consider the following definition of the tuple a_person; shown in Figure 2. The field name is represented as an offset from the start of the memory address for the tuple. In Figure 2, the name field is 8 bytes from the start of the tuple, as integers and reals take up 4 butes of memory.

```
var a_person : tuple
    age : INTEGER
    height : REAL
    name: STRING
    end
```

The logical objects you can reference are: a_person.age a_person.height a_person.name

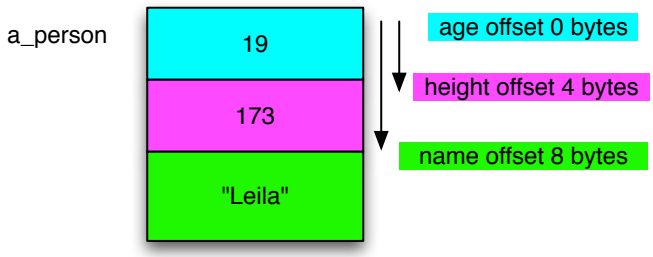

Figure 2: Example of the tuple a_person.

Mathematically a tuple is denoted as a cross−product of the tuple field types.

```
tuple_name : field_type_1 × … × field_type_n
```

The example tuple a_person would be denoted mathematically as
a_person : INTEGER × REAL × STRING.
Selecting a field mathematically is denoted by field_name(tuple_name). In our example the field values are denoted using age(a_person) and height(a_person) and name(a_person), because accessing the field values is invoking a parameterless function on the object tuple.

4 Operations on tuples

The operations are described using procedures and functions. However, because the tuple data type is built into many programming languages, a different special purpose syntax is used within such programming languages when tuples are referenced – see the section *The tuple fields*.

4.1 Enquiry operations

Although the implementation of tuples requires a tuple descriptor which is analogous to an array descriptor – in that it is used to map logical names to physical addresses – it is rare to have enquiry operations on tuple descriptors.

4.2 Read operations

As for arrays, we really only have one operation and that is to retrieve the address of any field in a tuple. The functional definition of this operation is analogous to that for arrays where the field_name in a tuple corresponds to an index for an array.

4.2.1 What is the address of a specified tuple field?

```
function address( a_tuple_name : NAME ; a_field_name : NAME) : REFERENCE
```

require a_tuple_name ≠ void **and** a_field_name ≠ void
ensure Result = address(a_tuple_name) + offset(a_field_name)

Program text is not referenced

4.3 Write operations

These contain the basic create_tuple and delete_tuple operations that are required for any data structure.

4.3.1 Create a new tuple

```
create_tuple(a_tuple_name : NAME ; ... ) : REFERENCE
```

require a_tuple_name = void
ensure Result = address(a_tuple_name) –– reference to a tuple with space sufficient to store the specified fields.

We will not attempt to define the parameters to the procedure any further because of the complexity of the parameter list. Basically all the data in a tuple definition are the parameters to the create operation.

Program text is not referenced

4.3.2 Dispose of an existing tuple

```
procedure delete_tuple( a_tuple_name : NAME)
```

require a_tuple_name ≠ void **and**
ensure a_tuple_name is no longer a valid name as the array is removed from the environment.

Program text is not referenced

5 Storage allocation and variant tuples

The memory space allocated to each tuple consists of sufficient space to store each field. Common practice is to store each of the tuple fields in adjacent memory locations in the order in which they are defined within the tuple definition; see Figure 2.

Each tuple type has associated with it a tuple descriptor. The tuple descriptor contains a list of the fields, their sizes and relative offsets from the beginning of the memory space

allocated to the tuple. When a field is referenced, the tuple descriptor is found and the appropriate offset is added to the starting location for the tuple; see the section *What is the address of a specified tuple field?*.

With tuple structures, it is common to permit variant tuples; see Figure 3. These are tuples which have varying structure depending upon the value in a tag field. The usual physical representation is to have each tuple occupy an amount of space which is large enough to hold the variant requiring the greatest amount of space. Then the variable names associated with each variant map to the appropriate memory location and the data within that location is further interpreted according to the type of the variant field.

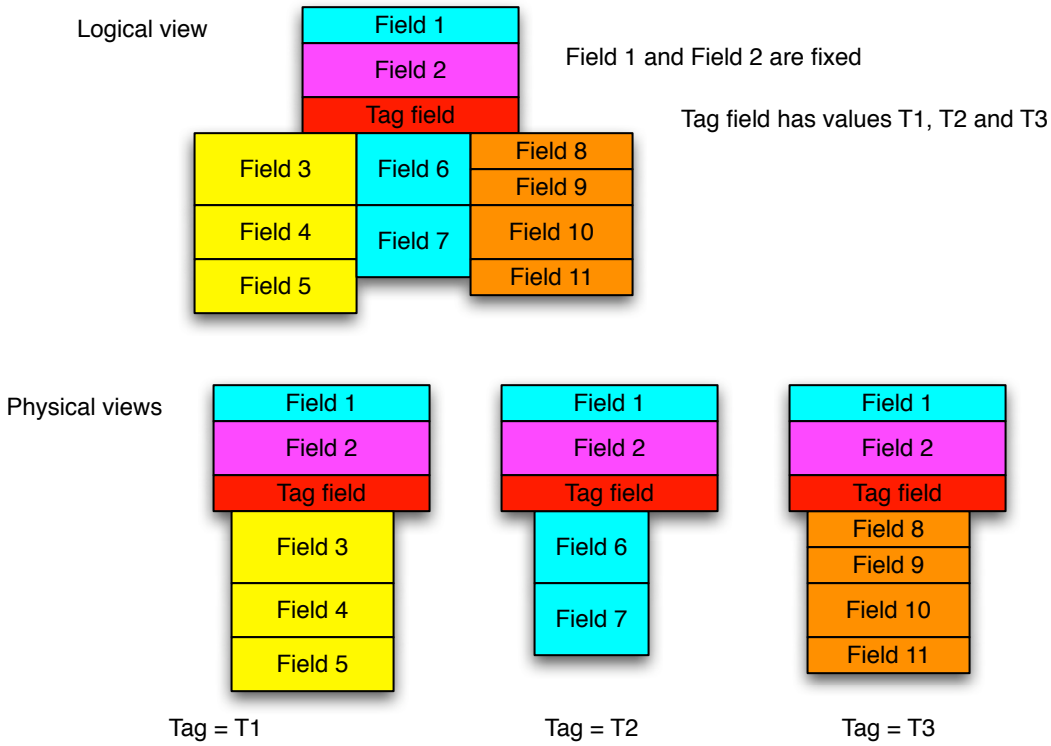

Figure 3: An example of a variant tuple.

Most often a variant tuple has all of its variants at the end of the tuple with a, possibly empty, fixed portion at the beginning of the tuple; see Figure 3. There is no logical necessity, in general, for this standard but it is the simplest to implement. It was introduced by Wirth in the Pascal language as a means to minimize potential programming errors. Normal practice, for example, would have a programmer define a tag field in the fixed part of a variant tuple and, depending upon the value of the tag, the programmer would branch to different sections of the program, where each such section would reference different variant fields.

Explicitly storing a tag field that defines the variant in effect, is recommended, as then the programmer can check the type to see that the correct variant field is being used. Since there is no run time checking for the variant type, the programmer can still make errors, simply by storing the wrong type information or referencing invalid fields for the tuple type.

In conclusion, with a tuple structure we must store the entire function table that maps field names to relative offsets in the tuple descriptor, as we cannot compute the location of arbitrary elements, as can be done with an array. We must do a table look up to find the relative position of an element.