

# Advanced Programming

## final term paper

Cosimo Sacco

### Abstract

This paper describes the *PAQL code generator*, a *Java* and *C++* project developed by the author in the frame of the *Advanced Programming* academic course. *PAQL* is an *element*, *container* and *query* description language. The *PAQL code generator* translates a *PAQL description* into a suitable set of *C++* classes simulating the *LINQ C# mechanism*.

## 1 PAQL basic grammar

*PAQL* basic grammar  $G_{PAQL}$ :

```
 $\langle description \rangle ::= \langle element \rangle \langle description \rangle$   
|  $\langle container \rangle \langle description \rangle$   
|  $\langle query \rangle \langle description \rangle$   
|  $\varepsilon$   
  
 $\langle element \rangle ::= \text{'element' } \langle identifier \rangle \text{'{' } \langle declarationBlock \rangle \text{'}'}$   
  
 $\langle declarationBlock \rangle ::= \langle keyDeclaration \rangle \langle declarationBlock \rangle$   
|  $\text{'{' } \langle varDeclarationBlock \rangle \text{'}'}$   
  
 $\langle keyDeclaration \rangle ::= \text{'key' } \text{'{' } \langle varDeclarationBlock \rangle \text{'}'}$   
  
 $\langle varDeclarationBlock \rangle ::= \langle variableDeclaration \rangle \langle varDeclarationBlock \rangle$   
|  $\varepsilon$   
  
 $\langle variableDeclaration \rangle ::= \langle identifier \rangle \langle identifier \rangle \text{';'}$   
  
 $\langle container \rangle ::= \text{'container' } \text{'<' } \langle identifier \rangle \text{'>' } \langle identifier \rangle \text{';'}$   
  
 $\langle query \rangle ::= \text{'query' } \text{'<' } \langle identifier \rangle \text{'>' } \langle identifier \rangle \text{'(' } \langle identifier \rangle \text{' )' ';'}$ 
```

This grammar allows *multiple superkeys with variable arity*. The related parsing table  $P$  follows

	$\varepsilon$	identifier	query	container	}	{	key	element
$\langle \text{description} \rangle$	$a_3$		$a_2$	$a_1$				$a_0$
$\langle \text{element} \rangle$								$b$
$\langle \text{declarationBlock} \rangle$						$c_1$	$c_0$	
$\langle \text{keyDeclaration} \rangle$							$d$	
$\langle \text{varDeclarationBlock} \rangle$		$e_0$			$e_1$			
$\langle \text{varDeclaration} \rangle$		$f$						
$\langle \text{container} \rangle$				$g$				
$\langle \text{query} \rangle$			$h$					

Let  $c_{i,j}$  be the cell of  $P$  in the position  $(i,j)$ . Then

$$\#(c_{i,j}) \in \{0,1\} \quad \forall i,j : c_{i,j} \in P \implies G_{PAQL} \sim LL(1).$$

## 2 Generated classes

### 2.1 Element model class

The element model class is a simple struct which defines the *equal to* operator as follows.

```
bool operator==(const Element& e) const
{
    return (e.v1 == v1) && ... && (e.vN == vN) && (true);
}
```

Where  $v_i$  represents the  $i$ -th declared variable.

### 2.2 Container model class

This *PAQL* implementation allows *multiple superkeys with variable arity and composing types*. Each key is defined by a tuple of types. An example:

```
key {int anagraphicCode; string name;}
key {int code; string nation;}
key {Genoma personalSequence;}
```

In the example, different keys have the same type tuple.

In order to provide an efficient container lookup, a *map* should be provided for each defined key, and a proper *get* method which uses the right map to find the result. The main problems to face in this case are three:

1. The “*key type*” is a type tuple. How can a tuple of hetherogeneous types be managed?
2. The generated methods `Element& get(int anagrCode, string name)` and `Element& get(int code, string nation)` would have the same signature. How can two key types whose type tuple is the same be distinguished?

3. The `Element& get(int anagrCode, string name)` and `Element& get(int code, string nation)` methods should lookup on two different maps. How can a key type be linked to the proper map?

Those problems are addressed through *template metaprogramming* techniques. A container is implemented as the *fusion* of a *generic list* and a *meta map*<sup>1</sup> through *multiple virtual inheritance* from those structures<sup>2</sup>. The “*list nature*” of the container allows general element access, while its “*meta map nature*” conveys to fast access. Figure 1 shows the type relations involved in each container class. A `Container` is a list of references<sup>3</sup> to elements and a *set of maps*

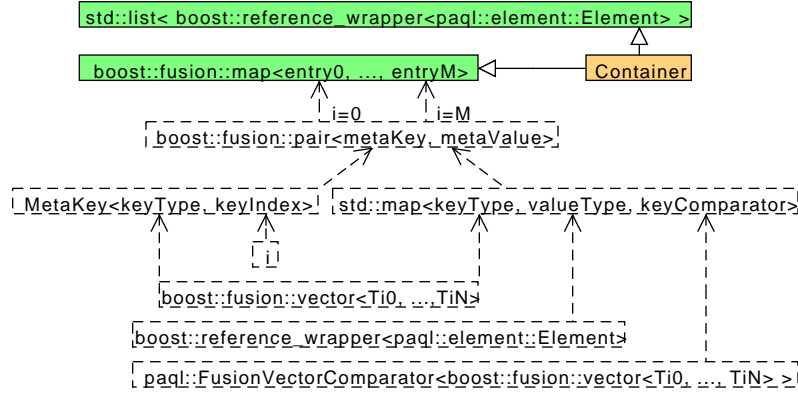


Figure 1: The container type genesis

which index the contained elements according to each key definition. Each map is identified by its *meta key*, which is a type, and is retrieved at compile time through the *meta selection operator* `boost::fusion::at_key<keyType>`. The *meta keys* are composed by the *key type*, which is a *type tuple* composed through the *variadic template class* `boost::fusion::vector<T0, ..., TN>`, and its index, which is an integer constant. Each retrieved map is indexed by the related *key type*, and refers the same objects contained in the references list. The operations defined on a container are:

- **insertion:** given an element to insert, a new entry is created in each map and gets indexed by the proper key (created using, each time, the proper element members) and the element reference is added to the list;
- **selection:** given a key of some type, a proper *get* method is provided, which selects the element reference using the passed key in the right map;

<sup>1</sup>A *meta map* is a map where the keys are types, and access is resolved at compile time.

<sup>2</sup>Implemented by `std::list` and `boost::fusion::map` respectively.

<sup>3</sup>`boost::reference_wrapper` allows to pass references as template parameters.

- **removal**: given an element reference, the *remove* to delete each entry related to that element in each map, and remove it from the list;

An example of a container class can be found in `paql/test/paqlClasses/ContainerExample`.

## 2.3 Query model class

In the first, basic version of *PAQL*, the generated query classes are very simple. The query execution method simply retrieves all the elements from its container instance `c` and returns the formed list.

```
std::list<boost::reference_wrapper<paql::element::Element>> execute()
{
    typedef std::list<boost::reference_wrapper<paql::element::Element>> List;
    List r;
    for(List::iterator i = c.begin(); i != c.end(); i++) r.push_back((*i));
    return r;
}
```

## 3 Parsing and code generation

A **System** is an object which transforms a generic *input* object into an *output* object through its `transform(Input, Output)` method. A **Compiler** is a **System** which transforms an `InputStream` into a `List<SourceFile>`. A **Compiler** is made up of several `subSystem`, linked in a *pipeline* where each *output* is an *input* for the following. Each subsystem manages a particular phase of the compilation process, in particular

- **LexicalAnalyzer**: given an *input stream*, it recognizes valid *language tokens*, thus performing a *lexical check*, and builds up a *token list*;
- **SyntacticAnalyzer**: given a *list of valid tokens*, it recognizes valid *syntactic structures* and represents them in the form of a *parse tree*, thus performing a *syntactic check*;
- **SemanticAnalyzer**: given a valid *parse tree*, it checks its semantic validity and builds up *language specific semantic structures*;
- **CodeGenerator**: given a valid *semantic structure*, it serializes it to a *list of source files* whose language is the *object language*;

Those interfaces are implemented to satisfy the *PAQL* specifications, so the *PAQL* compiler is built up by the systems of concrete classes `PAQLLexicalAnalyzer`, `PAQLSyntacticAnalyzer`, `PAQLSemanticAnalyzer` and `PAQLCodeGenerator`. Figure 2 shows the class hierarchy and relations. *Tokens*, *parse trees*, *semantic structures* and *source files* are represented by dedicated classes. Figure 3 shows the class hierarchy of tokens and parse trees. In particular, both `Token` and



