Building Functional Techniques into an Object-oriented System

Elena BOLSHAKOVA, Andrey STOLYAROV

This is the draft version of the paper which had been presented at the 4th Joint Conference for Knowledge-Based Software Engeneering at Brno, in September 2000. The paper is published in the proceedings and that publishewd version should always be cited in preference to this draft using the following citation details:

Elena Bolshakova and Andrey Stolyarov. Building functional techniques into an object-oriented system. In *Knowledge-Based Software Engineering. Proceedings of the 4th JCKBSE*, volume 62 of *Frontiers in Artificial Intelligence and Applications*, pages 101–106, Brno, Czech Republic, September 2000. IOS Press, Amsterdam, 2000.

The appropriate LaTeXbibitem follows:

```
@INPROCEEDINGS{bolshstol:building,
        AUTHOR = "Elena Bolshakova and Andrey Stolyarov",
        TITLE = "Building functional techniques
                 into an object-oriented system",
        BOOKTITLE = "Knowledge-Based Software Engineering.
                    Proceedings of the 4th JCKBSE",
        ADDRESS = "Brno, Czech Republic",
        PAGES = "101-106",
        PUBLISHER = "IOS Press, Amsterdam",
        SERIES = "Frontiers in Artificial
                  Intelligence and Applications",
        VOLUME = {62},
        MONTH = "September",
        YEAR = {2000},
}
```

Building Functional Techniques into an Object-oriented System

Elena BOLSHAKOVA, Andrey STOLYAROV
Lomonossov's Moscow State University,
dept. of Computational Mathematics and Cybernetics,
Vorobjevy Gory, 119899, Moscow, Russia.

Abstract.
A method is proposed how to integrate techniques developed within various programming styles (paradigms) on the basis of object-oriented style, permitting to keep the significant facilities of programming languages to be integrated.

The paper starts with a brief description and comparison of basic programming paradigms (i.e., imperative, functional, logic and object-oriented) and corresponding programming techniques. Then it is argued that integration of different techniques is needed to simplify implementation of large programming projects. The key idea of the integration method proposed is to build into widespread imperative object-oriented language (such as C++ or Ada) useful techniques from another paradigm via creating in the former language a set of special classes of objects and operations. These classes represent a knowledge about syntax and semantics of language constructs to be built into. It is considered important that the source object-oriented program code with the built-in constructions would be compiled in an ordinary manner without any preprocessing, and that there's no syntactic analysis of the code at the runtime.

The paper describes in more details the experimental implementation of the method discussed: functional techniques of the programming language Lisp (including Lisp structures and functionals) were built into the C++ language.

1 Introduction

Modern programming is based on several main styles (paradigms, or models) of programming - imperative, functional, logic and object-oriented styles, which present different modes of thinking about a program and its execution. Various programming techniques (data and control structures, language constructs, etc.) were developed independently within each paradigm during the more than forty-years history of programming, but nowadays the necessity of integration of these techniques is realizable. Though algorithmic languages usually include programming techniques from different styles, they may be classified in accordance with the main style and techniques supported.

The imperative (procedural) style is the oldest and most traditional. It has grown from machine and assembler languages, whose main features reflect the John von Neuman principles of computer architecture. Thus, the imperative program consists of explicit commands (instructions) to be consequently executed carrying out operations on data and modifying the values of program variables and external environment.

The functional style is in fact old style too, since it has arisen from evaluation of algebraic formulae and its elements were used in first imperative algorithmic languages, such as Fortran. The pure functional program is a collection of mutually related (and possibly recursive) functions, each of them represents a certain expression for computing a value. Execution of functional program is simply evaluation of these values.

Within the logic paradigm the program is thought of as a set of logic formulae - axioms (facts), describing properties of certain objects, and a theorem to be proven. Program execution is the logical proving (inference) of the theorem, and successful proving results in construction of the objects with properties described.

The essential difference between these three styles concerns not only the concept of program and its execution, but also the concept of program variable. In contrast with imperative programs, there are neither explicit assignment statements nor side effects in the pure functional and logical programs. The variables in such a program are similar to those in mathematics - they denote actual values of function arguments or objects being constructed during inference. Unlike imperative languages, functional languages (Lisp [6], Refal [8], etc.[2]) are usually recursive and interpretive, most of them are oriented towards symbolic processing, as well as the unique famous logic language Prolog [5].

Programming techniques developed within functional and logic styles include (besides recursion) flexible data structures for representing complex symbolic data (Lisp, Refal, Prolog), pattern matching mechanisms (Refal, Prolog), functionals, or high order functions (Lisp), and mechanisms of partial evaluations (Refal).

In the object-oriented style a program describes the structure and behavior of so called objects and classes of objects. Classes represent sets of objects with the same structure and behavior (operations, or methods). In general case descriptions of classes compose the inheritance hierarchy including polymorphism of operations. Execution of object-oriented program is regarded as exchange of messages between objects modifying their states.

The object-oriented paradigm is the most abstract, its basic ideas can be easily combined with principles and programming techniques of other styles. Really, an object method may be interpreted as a procedure or a function, whereas the sending of message may be interpreted as procedure or function call. Just that way of integration of basic object principles with programming techniques of different styles has led to object-oriented variants of famous languages, for example, C++ [7], Ada95 [4] or Object Prolog.

Each programming style has its own scope of applicability: functional programming is useful for symbolic processing, while logic programming is useful for deductive data bases and expert systems, but both of them are not suitable for interactive tasks. Imperative languages are equally convenient for numeric and symbolic computations, giving up to most of the functional languages (and Prolog) in the power of symbolic processing techniques. The object paradigm is useful for creating large programs (especially interactive) with complicated behavior and with various types of data.

2 The key idea

Nowadays, imperative object-oriented languages C++, Ada and Object Pascal are the most popular for implementation of large program project. But in large projects one or several subproblems often belong to the symbolic processing domain, for which functional languages are more suitable. For example, creating a complicated data base (of a hundred or so various relations) with a natural language interface involves the task of syntax and semantic analysis of natural language queries, including elements of logical inference. Hence, languages such as Lisp or Prolog are preferable for this task. We see that the integration of programming techniques from different programming styles is needed to simplify the implementation of large programming projects.

It seems quite attractive to enhance the power of popular imperative object-oriented languages with special data structures and control mechanisms from non-traditional languages such as Lisp and Prolog. We'll discuss a method how to integrate techniques developed within various programming styles on the bases of object-oriented paradigm. The key idea of the method is to build into imperative object-oriented language useful techniques from another style (language) via creating in the former language a set of special classes of objects and operations. These classes represent a knowledge about syntax and semantics of language constructions being built in, and permit to model these constructs within the object-oriented language. It is assumed that this object-oriented language allows overriding of standard operations for user-defined classes (types), as the language C++ or Ada do.

Such a way of integration makes it possible to use different styles and techniques of programming within the same file (program module) working (in a strict sence) with the object-oriented programming language only. One might call this way as "immediate integration", though built-in techniques and constructions may have modified syntax (but not semantics). It is considered important that the source object-oriented program code with the built-in constructions would be compiled in an ordinary manner without any preprocessing, and that there is no syntactic analysis of the code at the runtime.

3 Experimental implementation overview

An experimental implementation of the integration method discussed has been carried out for building functional techniques of programming language Lisp into C++ language resulting in the InteLib library of C++ classes. This library supports a dialect of Lisp with the following facilities:

- Lisp S-expressions (numberic and string constants, symbols, dot pairs and lists);
- Wide range of built-in functions and special forms, starting with CAR and CDR, ending with MAPCAR, APPLY, EVAL;
- User-defined functions (lexical closures), lambda-expressions;
- User-defined macros (in the sence of Common Lisp [6]) and so called nlambda functions (functions that don't evaluate arguments).

In contrast with most of known dialects, in the dialect supported by InteLib (InteLib Lisp), one can pass to a functional (such as MAPCAR) as its argument not only an ordinary function, but also any functional object such as nlambda functions, special forms and even macros.

Both lexical and dynamic styles of bindings of Lisp variables (symbols) are available as well as supporting constant symbols. InteLib provides DEFVAR and DEFCONST special forms (similar to those in Common Lisp) for this purpose. The library also implements a collection of miscellaneous functions (such as input/output functions, hash table handling functions and so on).

4 Architecture and implementation of the InteLib library

InteLib is based on a hierarchy of polymorphic classes representing various types of S-expressions. The abstract base class LTerm implements the common properties of all these types. LTerm has a method Evaluate intended to evaluate and return the value (in the sence of Lisp) of the given S-expression.

In the discussed version of the library LTerm class has the following child classes:

- LTermNil (represents empty list);

- LTermInteger and LTermFloat (numberic constants);
- LTermString (string constants);
- LTermSymbol (Lisp symbols);
- LDotPair (elements of a list).
- LForm and its child classes corresponding to various functional forms.

- $LTerm$
  - $LTermInteger$
  - $LTermFloat$
  - $LTermString$
  - $LTermNil$
  - $LTermSymbol$
  - $LDotPair$
  - $LForm$
    - $LCFunction$
      - $...All\, Functions\, implemented\, in\, C++$
    - $LLispForm$
      - $LLambda$
      - $LNLambda$
      - $LMacro$
    - $...All\, special\, forms$
  - $LTermStream$
  - $LHashTable$
  - ...

The LTermSymbol class shown at the diagram in fact stores a reference to an object of LSymbol class, which is not a member of the LTerm hierarchy itself. LSymbol class stores global (dynamic) value association, function association, text representation of the symbol and binding scheme (either lexical or dynamic).

To represent functionals as data, LForm subhierarchy has been developed within the LTerm hierarchy. The subhierarchy has LCFunction and LLispForm clases. Lisp special forms are represented with classes also descended directly from LForm.

The LCFunction class represents functions implemented in C++, including all "built-in" functions such as CAR, CDR, CONS etc. To add a new Lisp function to the library, a programmer needs to declare a class descended from LCFunction with only one method (DoCall) overriden.

The LLispForm class is intended to represent forms defined with Lisp constructions (lambda functions, nlambda functions and macros). It has references to the lambda-list (the list of formal parameters), the function body and the lexical context of the form. LLispForm class has child classes corresponding to different kinds of forms:

- LLambda (ordinary Lisp function);
- LMacro (Lisp macro evaluated as in Common Lisp);
- LNLambda (Lisp function that does not evaluate its arguments).

In most cases, the objects of LTerm class hierarchy reside in dynamic memory and are not operated directly (altough it is possible). In order to provide garbage collection, a class LReference has been created. LReference behaves like a pointer to LTerm and notifies the pointed object that a new pointer is created or that an existing pointer is either destroyed or reassigned. Using these notifications, LTerm object counts existing references to itself and deletes itself when there is no more references.

LReference class is designed to be primary API to Lisp objects represented with InteLib: it has about 20 constructors that allow to create various types of Lisp objects. In particular, it is possible to construct a Lisp object from any of standard C numeric or string constant. It is also possible to create an LTerm object explicitly with operator new and let LReference take care about it.

To construct Lisp lists and to specify them within C++ programs, special class LListConstructor has been added to InteLib and a C++ built-in operation "," (comma) has been overriden for LReference class as the left operand. LListConstructor class has no data members and usually has only one instance named L. Here's its complete definition:

```
 class LListConstructor { public:
   template <class T> LReference operator |(T t)
     { return LReference(new LDotPair(LReference(t))); }};
```

The only purpose of this class is to turn anything into a Lisp list of one member.

The new version of comma operation assumes its left-hand operand is an LReference object pointing to a list (possibly empty) and appends its right-hand operand to this list. C++ implicit type conversion allows to use any numeric and string constants as a right-hand operand of the comma operation as well as LSymbol class objects.

Besides that, "~" (tilde) operation has been overriden for LReference and LSymbol classes to represent abbreviated form of QUOTE traditional for Lisp.

Using LListConstructor and the two operations with the new semantics, a programmer can represent Lisp lists in C++ code like this:

```
(L| "I am the walrus", 1965)    // ("I am the walrus" 1965)
(L| 1, 2, (L| 3, 4), 5, 6)      // (1 2 (3 4) 5 6)
(L| (L| 1, 2), 3, 4)            // ((1 2) 3 4)
(L| CAR, ~(L| 1, 2))            // (CAR '(1 2))
```

To the right of comment symbol, the represented lists are written in traditional Lisp syntax.

The syntax shown is clear enough for a human programmer to work with, but in case of large fragments of code to be written in Lisp it might be more convenient to code in traditional syntax. The process of translation from traditional Lisp into the corresponding C++ constructions is quite simple and easy to implement. A translator has been created for this purpose.

5 One example

In conclusion we give an example of Lisp function in traditional syntax and its C++ analogue. Function ISOMORPHIC takes two trees as arguments and determines whether they have the same topology (equivalent structures):

```
(defun isomorphic (tree1 tree2)
   (cond ((atom tree1) (atom tree2))
         ((atom tree2) NIL)
         (t (and (isomorphic (car tree1) (car tree2))
```

```
                    (isomorphic (cdr tree1) (cdr tree2))))))
```

The result of translation of this code into C++ language generated by the translator follows:

```
LSymbol ISOMORPHIC("ISOMORPHIC");
LSymbol TREE1("TREE1");
LSymbol TREE2("TREE2");
static void LispInit() {
  (L|DEFUN, ISOMORPHIC, (L|TREE1, TREE2),
    (L|COND,
      (L|(L|ATOM, TREE1), (L|ATOM, TREE2)),
      (L|(L|ATOM, TREE2), NIL),
      (L|T, (L|AND,
        (L|ISOMORPHIC, (L|CAR, TREE1), (L|CAR, TREE2)),
        (L|ISOMORPHIC, (L|CDR, TREE1), (L|CDR, TREE2))))))
  .Evaluate();
}
```

Please note that symbols that correspond to standard Lisp functions used in the example (DEFUN, COND, ATOM, NIL, T, AND, CAR and CDR) are provided by the library so there's no need to define them explicitly.

In order to use this code, the programmer must call the function LispInit which evaluates the only top-level form of the example (DEFUN special form). Then the function ISOMORPHIC can be used in the C++ program.

6 Conclusion

In fact, the idea of translation of Lisp code into an imperative language (primarily into C) is not new (e.g., Gambit-C [1]). The main advantage of the proposed method is that the C++ code that corresponds to a Lisp code remains quite human-readable and human-maintainable after translation. Moreover, the use of translator is not necessary, one can write directly in C++ using Lisp's technique.

Now the ways of further development of the library are investigated. The set of classes representing semantics of Refal language kernel is being created on the bases of InteLib. Also we plan to build some dialect of Prolog into C++ language using this method.

References

[1] Feeley M. Gambit Scheme. http://www.iro.umontreal.ca/˜gambit/

[2] Field A., Harrison P. Functional Programming. Adison-Wesley, 1988.

[3] McCarthy J. Recursive functions of symbolic expressions and their computation by machine. Communications of the ACM, 3 (1960), 184-195

[4] Mitchell R. Abstract Data Types and Ada. Prentice Hall, 1996.

[5] Robinson J. Logic programming - past, present and future. New Generation Computing, 1, 1983, p.107-121.

[6] Steele G. L. Common Lisp the Language, 2nd edition. Digital Press, 1990.

[7] Stroustrup B. The C++ Programming Language, 3rd edition. Addison-Wesley. 1997.

[8] Turchin V. REFAL-5, Programming Guide and Reference Manual. New England Publishing Co., Holyoke, 1989.