A framework of heterogenous dynamic data structures for object-oriented environment: the S-expression model

Andrey V. STOLYAROV

Copyright ©Andrey V. Stolyarov, 2004.

This is the draft version of the paper which had been presented at the 6th Joint Conference on Knowledge-Based Software Engeneering at Protvino, Russia, at August 25, 2004. The paper is published in the proceedings and that published version should always be cited in preference to this draft using the following citation details:

Andrey V. Stolyarov. **A framework of heterogenous dynamic data structures for object-oriented environment: the S-expression model.** In *V. Stefanuk, K. Kaijiri, eds., Knowledge-Based Software Engineering. Proceedings of the 6th JCKBSE*, volume 108 of *Frontiers in Artificial Intelligence and Applications*, pages 75–82, Protvino, Russia, August 2004. IOS Press.

The appropriate LaTeXbibitem follows:

```
@INPROCEEDINGS{stolyarov:heterogenous,
        AUTHOR = "Andrey V. Stolyarov",
        TITLE = "A framework of heterogenous dynamic
                data structures for object-oriented
                environment: the S-expression model",
        EDITOR = "Vadim Stefanuk and Kenji Kaijiri"
        BOOKTITLE = "Knowledge-Based Software Engineering.
                    Proceedings of the 6th JCKBSE",
        ADDRESS = "Protvino, Russia",
        PAGES = "101--106",
        PUBLISHER = "IOS Press",
        SERIES = "Frontiers in Artificial
                  Intelligence and Applications",
        VOLUME = {108},
        MONTH = "August",
        YEAR = {2004},
}
```

A framework of heterogenous dynamic data structures for object-oriented environment: the S-expression model

Andrey V. STOLYAROV
Moscow Lomonossov State University,
dept. of Computational Mathematics and Cybernetics,
kom.747, str.52, vl.1, Leninskie Gory, 119899, Moscow, Russia.
e-mail: avst@cs.msu.ru

Abstract. There are various problem domains (including symbolic computations, computer algebra and others) in which it is convenient to use typeless programming languages such as Lisp, Refal, Prolog etc. It is learned from practice that the importance of the typelessness of these languages is sometimes no less than the importance of their primary programming paradigms.

In the paper the well-known S-expression model of heterogenous (mixed type) dynamic data structures is considered as a separate programming paradigm which is to be imported into an object-oriented environment, namely the environment of the C++ language. A library of C++ classes and templates which allows to handle S-expressions within a C++ program is described. The library lets the programmer to use natural Lisp-like syntax creating S-expressions. Using S-expressions as containers for objects of user-invented data types is also possible.

1 Introduction

Nowadays, imperative object-oriented languages such as C++[1], Java, C#, etc. are popular in the software industry. While many programmers are able to use programming paradigms other than imperative and object-oriented, and in almost any project there are subtasks where alternate paradigms are useful, there's often no chance to use paradigms such as functional programming [2] in a real project due to numerous difficulties with integrating different programming languages within a single project.

The solution first introduced in [3] is to create a class library for a popular language such as C++ which implements a computation model together with data structures of an alternate programming language. The InteLib library as introduced in [3] was capable to let the programmer do Lisp programming within a C++ program using expressions syntactically similar to traditional S-expressions [4].

The library was later extended for the computational model of Refal [5]. A demo version for a small subset of Prolog [6] was also created. The results are described in [9]. All the parts of the library use the data structures initially created for the Lisp part, which in fact just models S-expressions[1].

[1]The Refal part performs actual computations on certain optimized data structures, so the S-expressions are used only to build functions and the input data for them.

The notion of S-expression plays a significant role in what we know as ”Lisp programming”. Practice shows that it is actually possible to make use of S-expressions without the Lisp itself (and without the notion of *evaluation* of S-expressions). Other programming languages, such as Refal or Prolog, in fact use similar models of data, inventing completely different evaluation model. The S-expression model may be used in traditional imperative programming as well; such data structures are good in situations when pieces of data of completely different nature are to be handled together (e.g. in formulae processing), or when it is hard to say what particular data will be used in a certain task (e.g. in rapid prototyping). Both cases do not imply (though not decline as well) the need of *evaluation* of the data structures in terms of Lisp or any special evaluation model.

This lets us to speak about S-expression model as a separate *programming paradigm* which doesn't depend on a particular evaluation model, including Lisp.

Once this fact was realized, the architecture of InteLib got refactored in order to separate the S-expressions themselves (as data structures) from the Lisp evaluator. In this paper the base part of the new version of InteLib is described. It is intended to let the programmer use heterogenous data structures within the C++ environment, even in the role of containers.

The base part of the new library enables a programmer to use the notion of S-expression (heterogenous data and conses as we used to use in Lisp) within a C++ project, in a natural way (with syntax which is close to the original Lisp), and without the need of any preprocessing of the code.

2 Architecture of the library

The library is organized as a core function set and a pool of optional features. The core provides classes which represent traditional atomic S-expressions and conses. The operations of constructing, handling and analysing Lisp-like lists are also a part of the core.

Simple reference counting mechanism of garbage collecting is provided; for the projects which use one of existing garbage collecting techniques, the reference counting can be switched off by a compile time option.

2.1 Primary types of S-expressions

The library provides the following basic atomic value types: integer constant, floating point number constant, string constant, and a label. Single character is considered to be equal to a string of one character.

The label is an S-expression for which only its identity is important, that is, each label differs from any other S-expression and that's all about it. Labels are known as *symbols* in Lisp, *constants* in Prolog and *label symbols* in Refal. Also this type of S-expressions is used for representing the *empty list* objects and the boolean *true* and *false* values.

The actual types of values stored by numberic S-expressions may be set to any of the built-in C types, e.g. to `long` for integer constants and `double` for floating point number constants; this is controlled by compile-time options.

Besides the atomic S-expressions, the core invents *conses*, the well-known type which consists of two references to other S-expressions. Having assigned an object of the type *label* to represent the empty list, one can build traditional Lisp lists which can consist of objects of different types including other lists.

The classes representing these S-expressions are named `SExpressionInt`, `SExpressionFloat`, `SExpressionString`, `SExpressionLabel` and `SExpressionCons`, respectively. All these classes are derived from the basic abstract class named `SExpression`. Unless disabled by a compile-time option, the class implements a mechanism of reference counting.

C++ constructs	Lisp equivalent
`(S| 25, 36, 49)`	`(25 36 49)`
`(S| "I am the walrus", 1965)`	`("I am the walrus" 1965)`
`(S| 1, 2, (S| 3, 4), 5, 6)`	`(1 2 (3 4) 5 6)`
`(S| (S| 1, 2), 3, 4)`	`((1 2) 3 4)`
`(S| 1 || 2)`	`(1 . 2)`
`((S| 1, 2, 3)|| 4)`	`(1 2 3 . 4)`

Table 1: Examples of S-expressions represented with C++ constructs

2.2 Common handler

In order to make handling of `SExpressions` more convenient, a "smart pointer" to an `SExpression` is made, named `SReference`. One of its primary duties is to perform reference counting on the `SExpression` objects; to do so, objects of the `SReference` class notify the respective `SExpression` objects about creating or removing a link to the object.

Besides that, `SReference` implements some generic support for the S-expressions included into the core part of the library. The class implements constructors for coercion from all existing C++ built-in numberic types (`char`, `short`, `int`, `long`, both signed and unsigned, and `float`, `double` and `long double`) as well as from `const char*` which stands for a string constant, creating S-expressions of the respective types[2]. A constructor for creating a cons is also available and it has two arguments of the `SReference` type, which are used as values of the *head* and *tail* parts of the newly-created cons. The class has methods `GetInteger()`, `GetFloat()` and `GetString()` which allow to convert the referenced S-expression into the respective built-in data type, if it is possible; when it is impossible, e.g. if `GetInteger()` is called for an object which references on a label S-expression, an exception is thrown. Functions `Car()` and `Cdr()` (which return the respective parts of a cons) are available to make list analysing easier. The functions `Car()` and `Cdr()` return a reference to the respective `SReference` inside the `SExpressionCons` object thus making possible changing the parts of the dot pair.

2.3 List composition operations

The authors of [8] offer to build lists of conses with a function `cons`, which builds a cons of the given two arguments. The function is called explicitly. This solves the problem of list building, but the form of the code doesn't seem to be similar to the traditional Lisp notation. E.g., the list `(1 2 3)` is to be represented with

```
cons(new l_int(1), cons(new l_int(2),
     cons(new l_int(3), &nil)));
```

where `l_int` is the class which represents Lisp integers, `nil` is the object which marks the list end, and `cons` is the functions which builds conses.

In practice such a notation is not appropriate because of its complexity and poor readability. Really, a programmer who is familiar with Lisp would hardly agree to use such unclear constructs to represent Lisp lists.

The `SReference` class implements a technique of constructing Lisp lists introduced in [3], which in fact solves the problem of complexity and clarity of list construction.

In Lisp there's an operation of constructing a list of an arbitrary length denoted by parentheses. The operation has variable 'arity'. For example, a construct `(1 2 3)` has 3 arguments and builds a

[2]Support of long long int depends on whether the compiler supports it.

list of 3 items - 1, 2 and 3. Besides that, there's a binary operation which builds a cons (dotted pair), such as `(1 . 2)`. The construct `(1 2 3)` has the same effect as a superposition of 3 dotted pair constructors, like this:

```
(1 2 3)    ==    (1 . (2 . (3 . NIL)))
```

One can implement an operation similar to the Lisp's '`(.)`' and it will allow us to build any list of S-expressions. It is though a bit inconvenient to create lists only using this operation (imagine you couldn't use plain lists in Lisp, only dotted pairs).

Another problem is that one might want to use just a C++ constant or expression without explicit cast of it to an S-expression, e.g. '3', not a construct like `SReference(3)`, so in case of an overloaded standard operation we need at least one of operands already of `SReference` type, which allow the compiler to understand we mean the overloaded operation, not the standard one. This requirement also prevents us from using C++ functions with unspecified number of arguments because there's no way to determine at run time what types of actual parameters do we have.

The problem is solved replacing the Lisp '`()`' operation of an arbitrary list composition with two operations. First of them creates a list of one element, while second adds an element to a given list. It is clear the two operations allow to create an arbitrary list, that is, their combination has the same functionality as the Lisp '`()`' operation.

The first operation always has exactly one argument, but we can't use a symbol of any standard unary operation for it because we want it to be applicable to an expression of some C++ built-in types. We also don't want to use a plain function for this purpose because parentheses would make our constructs less clear. The problem is solved with a class `SListConstructor`, which is created to be a label for a binary operation to show the compiler to apply an overloaded one instead of the built-in operation. Usually there's only one instance of `SListConstructor` named `S`. For example, an operator `S|3` returns an `SReference` object that represents Lisp construct `(3)`.

For appending a new item to a list we can overload any overloadable binary operator. For a better clarity, the C++ comma (`,`) is used for this purpose. Left-hand operand of a comma is always an `SReference` representing a list. Comma destructively changes the list replacing the final `NIL` with a dotted pair of `(X . NIL)` where `X` is its right-hand operand casted to an `SReference`. This makes it possible to represent Lisp lists in C++ as shown in table 1.

For composing dotted pairs and dotted lists InteLib offers an operation `||`. The left-side operand must be a list (possibly of one element). The operand appearing at the right side of the operator is converted to `SReference` and then the operation replaces the last `NIL` of the given list with whatever it constructed from the right-side operand. See table 1 for an example.

Note the parentheses in the last example. They appear because the comma operator has lower precedence than `||`.

2.4 Constant S-expression handling

For being consistent with C++ traditions, there is also a constant version of `SReference` named `SConstReference` which makes sure the referenced S-expression will not be changed unless implicit type conversion is used.

The `SConstReference` class has almost the same capabilities as ṠReference. except that some methods including `Car()` and `Cdr()` return constant references and pointers thus preventing the caller from changing any existing S-expressions. The list construction operations are also not available for `SConstReference` because of their destructive nature.

An object of `SConstReference` can be created as a copy of an `SReference` object. Attempt to make an `SReference` from `SConstReference` causes duplicating of the referenced data structure. Since some `SExpression` objects can not be modified anyway, only the objects of *changeable* S-expressions are copied. Among the mentioned types of S-expressions, only the

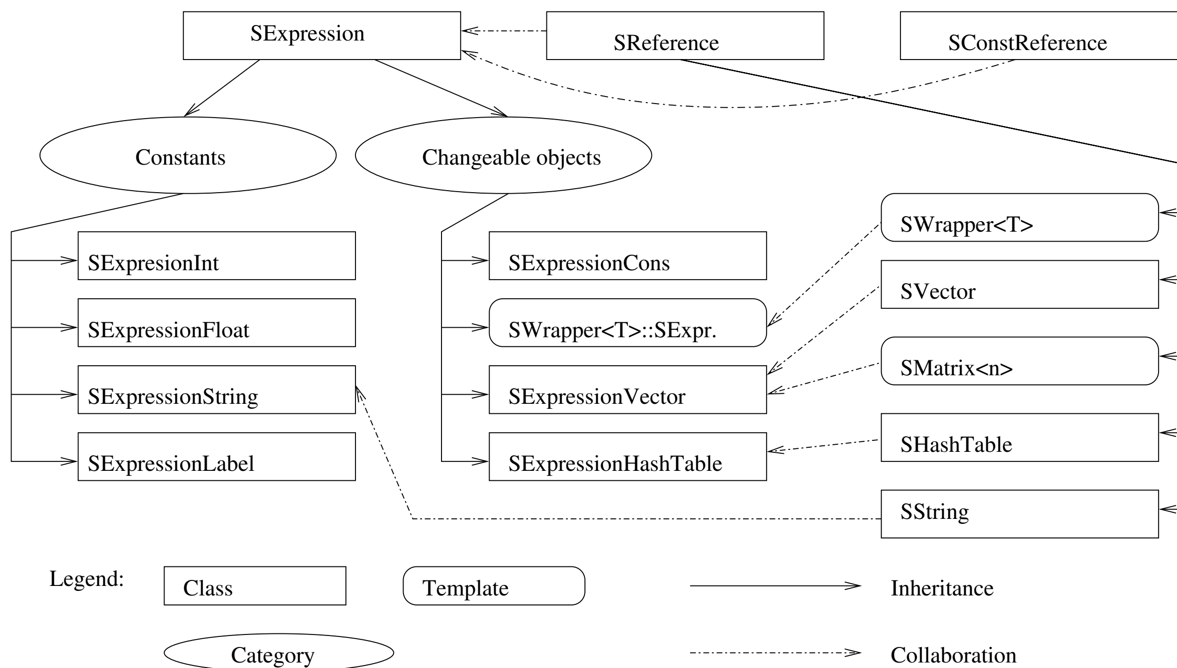

Figure 1: Classes hierarchy

`SExpressionCons` is considered changeable, while all the rest are not; that is, the process of duplication of an S-expression works recursively, so that the resulting copy has its own changeable items and shares those which cannot be changed with the source data structure. Note that a string constant is considered atomic and therefore unchangeable.

2.5 Additional data structures

Besides the classic S-expression types, the library also provides some additional facilities. First of them is the `SString` class derived from `SReference`, which always references to an object of the `SExpressionString` class and provides some obvious operations on strings, such as concatenation, inserting or erasing a sequence of characters etc. It is not so efficient as the well-known standard `string` class because it always creates a copy of the string on any modification. However, when being used for handling constant strings, it performs at the same level of efficiency as `string`. Using this class for internal purposes (namely for building and storing textual representations of S-expressions) allows not to depend on STL and, therefore, to use the library in projects where using of STL was declined for some reasons.

Another useful feature is a dynamically-resized vector of S-expressions. It is implemented by several classes. First of them, `SExpressionVector`, is a changeable S-expression which stores a vector of references to other S-expressions. Note that deriving this class from `SExpression` allows a vector to reference to other vectors emulating an arbitrary-dimensional array.

Another class, `SVectorRef`, derived from `SReference`, is an interface to vectors which provides `operator[]()` for users' convenience. This class is just a smart pointer to `SExpressionVector`, that is, when `SVectorRef` object is *assigned*, it leads to two references to the same vector. `SVectorRef` can also be a *null* reference. Therefore `SVectorRef` is not yet a "first-class container".

Behaviour reasonable for containers is implemented by `SVector` class. This class is derived from `SVectorRef` changing the functionality a bit. First, `SVector` is never a *null* reference; its constructor automatically creates an `SExpressionVector` object. Second difference is that `SVector` never shares the referenced object with others. Copies (clones) are made always when the `SVector`

is assigned to another vector in any way.

Besides that, there's also a template class `SMatrix<int dim>` for handling a matrix of S-expressions. The template's argument `dim` controls the returning type of the `operator[]()`: for `SMatrix<2>` it returns `SVector` object, and for any integral $n > 2$ `SMatrix<`n`>::operator[]()` returns an object of `SMatrix<`$n-1$`>`.

The class `SExpressionVector` has a virtual method which controls the ability of the vector to resize. By default, the method denies any resizing if the object was constructed with a particular size value; otherwise, if the object was constructed with its default constructor, the method resizes the actual vector to the closest power of 2 each time when an illegal (too high) index is passed to the `operator[]()`.

The last additional S-expression type to be mentioned is a hash table, which is implemented by `SExpressionHashTable`, `SHashTableRef` and `SHashTable` classes. Both the key and the value of an item stored in the table are S-expressions. The table always has a length of a prime number and resizes automatically as needed. The hash function is computed differently for different types of S-expressions using virtual methods specially intended for this purpose. The behaviour of `SHashTableRef` and `SHashTable` classes is similar to `SVectorRef` and `SVector`, respectively.

2.6 User-defined S-expressions

The library provides a template class named `SWrapper<class T>` which allows to create an S-expression of a user-defined type. The parameter `T` must be a structure or a class. `SWrapper` is derived from `SReference`, and it invents a member class `SWrapper<T>::SExpressionWrapper` which is derived from `SExpression`. `SWrapper<T>` acts like a pointer to an object of `T`, that is, it has the appropriate operators `*` and `->`. This allows S-expressions to be used as containers for data objects of arbitrary types. In contrast with traditional containers, this technique allows to store heterogenous data in a container, thus producing heterogenous lists, vectors, matrices and hash tables.

3 Conclusion

The library described in this paper was used as a basis for implementing computation models of Lisp and Refal as C++ class libraries; implementation of the Prolog and Datalog [7] models are in progress.

Also the library was used in some programs which has no direct relationship to multilingual programming. The list of these programs includes an object-oriented wrapper for the well-known XML[10] handling library Expat [11]. Actually, the XML data model is naturally isomorphic to the model of S-expressions, which makes it convenient to use S-expressions for accessing and analysing data extracted from an XML source.

Improving the C++ environment with the model of S-expressions can make it easier to solve in C++ some tasks which traditionally considered to be hard for C++ while easy for Lisp and other artificial intelligence languages. This also demonstrates the flexibility of C++ which in fact allows to do multiparadigm programming using only libraries, not language extensions.

Practice shows that the notion of S-expression, which initially came from Lisp programming, can as well be useful outsude the Lisp environment. This allows to reconsider a definition of Lisp programming from multiparadigm point of view; despite the fact that functional programming, including lambda calculus, is still important, sometimes Lisp programming can be considered as *programming with S-expressions*. This is perfectly illustrated by tasks of automated formulae processing (including symbolic differentiation); it is the notion of S-expression what makes Lisp specially useful in this problem domain.

Having said all this, we conclude the notion of S-expression may be considered as a separate programming paradigm which is not limited to Lisp environment. The library described in the paper

makes it possible to use this paradigm within the environment of C++, one of the popular industrial programming languages.

References

[1] Stroustrup B. The Design and Evolution of C++. Addison-Wesley, Reading, Massachusets, 1994.

[2] Field A., Harrison P. Functional Programming. Addison-Wesley, Reading, Massachusets, 1988.

[3] Bolshakova E., Stolyarov A. Building functional techniques into an object-oriented system. In: T.Hruska and M.Hashimoto, eds. Proceedings of the 4th JCKBSE, Brno, Chech Republic, 2000. p. 101-106. IOS Press, 2000.

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

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

[6] Calmerauer A., Kanoui H., and van Caneghem M. Prolog, bases théoriques et développements actuels. *Technique et Science Informatiques*, 2(4):271–311, 1983.

[7] Ceri S., Gottlob G., and Tanka L. Logic Programming and Databases. Springer-Verlag, Berlin, 1990.

[8] Tan K. S., Hans Steeb W., and Hardy Y.. Symbolic C++: An introduction to computer algebra using object-oriented programming. Springer, London, 2000.

[9] Stolyarov A. Integration of language mechanisms of different nature within a single programming language. Ph.D. (”candidate”) thesis. Moscow Lomonossov State University, Moscow, 2002. *in Russian*.

[10] Extensible Markup Language (XML). http://www.w3.org/XML

[11] Cooper C. Using Expat. 1999. www.xml.com/pub/1999/09/expat