A Simple Context-Based Markup Algorithm and its Efficient Implementation in CLOS

Roger Kehr

June 1997

Bericht TI-12/97
Institut für Theoretische Informatik
Fachbereich Informatik
Technische Hochschule Darmstadt

Abstract

This report describes a simple though powerful context-based markup algorithm and its implementation.

Markup algorithms are often used to tag data structures with markup for representation purposes. They often follow an event-dispatching mechanism that allows the declaration of markup tags depending on the context in which an event occurs.

We describe the markup algorithm which is used in the xindy index processor and show how the multi-method dispatching facilities of CLOS can be easily exploited to obtain an efficient implementation.

1 Introduction

It is often necessary to output some data structure stored in main memory into a data stream using some particular form of *representation*. Somehow this process can be seen as the reverse work of a *parser*. The reasons for representing data structures in a stream are often *persistence*, the need to store the data structures for later reuse, and *structured markup*, making the representation further available for other post-processing steps such as typesetting.

We are interested in the problem of *tagging* data to obtain a specific markup needed for typesetting purposes. This problem is easily solvable as long as the way how markup is assigned to data objects is fix. In this case the markup process can be directly coded into some algorithm.

If there is a need for *configurable markup*, the situation is slightly more complicated. For an illustration of the difference, consider a tree consisting of nodes of different types that have to be serialised into an output stream. Usually one traverses the tree and emits appropriate markup tags when moving from node to node. This kind of markup is called *environment-based markup*, since the objects (in this example the nodes of the tree) are encapsulated into an environment of markup tags.

In almost all cases the nodes of a particular type are tagged uniformly in the same manner. This kind of markup is often used to *dump* a representation of a tree into a data stream. However, there are application domains for which one wants to make the issued markup tags depending on the node's position within the tree.

Therefore, we can roughly divide the markup schemes into *context-free* and *context-based* (or *context-sensitive*) markup. In this paper we will further concentrate on context-based markup.

The rest of this paper is organised as follows. In section 2 we first discuss existing systems that already contributed solutions to this kind of problem. In section 3 we describe the application domain for which we needed a simpler solution to the problem. Finally we present our traversing algorithm and its efficient implementation in Clos [1, 3, 7].

2 Related Work

The problem of context-based markup has several interesting general solutions. We briefly present the solutions of two different systems: CoST [2] and Stil [6]. Both systems are intended as back-ends for SGML *parsers*. SGML documents must conform to a structure that is described in a BNF-like grammar, called the *document type definition* (DTD). A document conforming to some DTD can be represented as a tree having SGML *elements* as its nodes. A SGML parser validates a document

against its DTD and produces a structural description of its contents, the so-called ESIS (Element Structure Information Set).

Both systems read the output of a SGML-parser and construct an internal representation of the tree. This tree is then traversed in some order (typically preorder) and appropriate markup is output when entering or leaving a node. The *context* of a node is different in both systems. In CoST all nodes along the path from the root node to the current node form a node's context. In Stil the whole tree forms a node's context.

Both systems allow to specify markup depending on the particular context in a very powerful way. Each SGML element is represented as the instance of a particular class (CoST uses [incr Tcl] [5], Stil uses Clos as their underlying object systems). When a node is entered or left an event is generated. This essentially is a message sent to the current object whose class must offer an appropriate callback method.

The user of these systems simply binds methods to these classes that output the desired markup. Methods are written in the underlying language (Tcl in CoST, Clos in Stil). Both languages are extended by a rich set of primitives to make the context information available inside the method bodies.

Thus, one can implement very powerful context-based markup strategies using a mixture of declarative specifications and functional descriptions.

3 A Configurable Context-Based Markup Scheme

3.1 The Problem

When implementing the xindy index processor [4] we were faced with a similar though less complicated problem. After processing the index its internal tree representation has to be tagged with user definable markup.

The nodes of the tree are instances of internally defined classes as well as user-defined classes. An example of such user-defined classes are the so-called *location classes* that can be specified in the *index style*[1]. Location classes can be page numbers, section numbers, or other kinds of location references for which a class definition must be supplied in the index style. As the markup of different location classes may be different (e.g. page numbers are encapsulated into other markup than section numbers, or location references of the same class may need different markup depending on certain attribute values they own), we decided to offer to the users a context-based markup scheme to be sufficient powerful. Consider an index of a textbook that contains references to a location class of page numbers which has

[1] The index style contains a description of the occuring location classes and other objects.

to visually distinguish between definitions of items on a page and their usage. An example index is shown here:

search
    binary, **23**, 24–26
    sequential, **15**, 16, 31
    ordered, **18**, 19

Here all references to the definition of an item are typeset in boldface, whereas the references to the usage of an item are without any specific markup. This requires that the location references own attributes that can be used to distinguish their intended meaning. The attribute values can be accessed at tree traversal time.

For the intended application domain the functionality of systems such as Stil or CoST are not appropriate for two reasons: (1) the target group of users for an indexing system are typically non-programmers which cannot be expected to write Clos methods or Tcl code, and (2) the functionality offered by both systems is not necessary due to the limited context of an index.

Hence, we aimed for a more efficient solution to the markup problem. As a first restriction we decided to define only a subset of the elements along the path as a node's context. Furthermore we observed that in many cases there is a need for making the attributes of an instance accessible as context information.

After these simplification steps we ended up in a very simple model. The context information for a particular node did not exceed four different objects. The basic traversing order was preorder and events were generated when entering and leaving a node. For list nodes we additionally generated an event when moving from one element in the list to its successor.

3.2 An Efficient Implementation in CLOS

The implemented markup system consists of two parts, (1) the markup algorithm traversing the data structure and raising appropriate events, and (2) the user interface, establishing the corresponding event bindings.

We continue with the description of the underlying context-based markup scheme and the dispatching mechanism which is directly implemented using Clos method calls. Finally we describe the user interface responsible for the dynamic establishing of event bindings.

The tagging process directly operates on the tree representation as illustrated in figure 1. For each inner node there exists a path from the root to the current node. Starting from the root in figure 1, such a path could look like

index
  letter group = `"G"`
    index entry = `"foo"`
      location class = `"page-numbers"`
        attribute group = `1`
          location reference attribute = `"definition"`

The context of an object consists of this path and all attributes the object owns. In our system, an event is generated each time a node is entered for the first time or left forever. The event contains a certain subset of the information present in the current context and passes this information to the dispatching unit.

In contrast to STIL, which represents events as CLOS method calls using a *single-argument dispatch*, we use in xindy the *multi-method dispatch facilities* of CLOS to implement the event bindings. The current context of a node is entirely passed as the method's arguments to the multi-method dispatcher of CLOS that finds the appropriate method. In STIL there is only one method bound to a class and the method itself is responsible for issuing the desired markup. Hence, the user of STIL is forced to write a LISP-function that (a) accesses the necessary context of the node using functions, and (b) uses the results obtained from the context queries to output appropriate markup. This functional description of defining markup has been replaced in xindy by a pattern-oriented, purely declarative specification which is less powerful though in our opinion better understandable by the end-user.

The method definitions make heavy use of the `eql`-specialisation feature of method arguments. This feature specialises a method not only on a certain class, which is usually the case in traditional object-oriented languages, but rather specialises on a single instance which can be determined at run-time[2]. This allows to specialise an event-binding to only a particular subset of the context arguments.

A sample method that is called in response to an event generated when entering a location reference owning the attribute *definition*, belonging to class *pagenums*, and being positioned at an arbitrary depth is represented through the following method:

```
(defmethod do-markup-locref-open
    ((attr      (eql #<attribute "definition">))
     (locrefcls (eql #<class     "pagenums">))
     (depth     number))
  (do-markup-string "\macro{"))
```

[2]This is possible in CLOS because methods can be defined at run-time after objects have been instantiated.

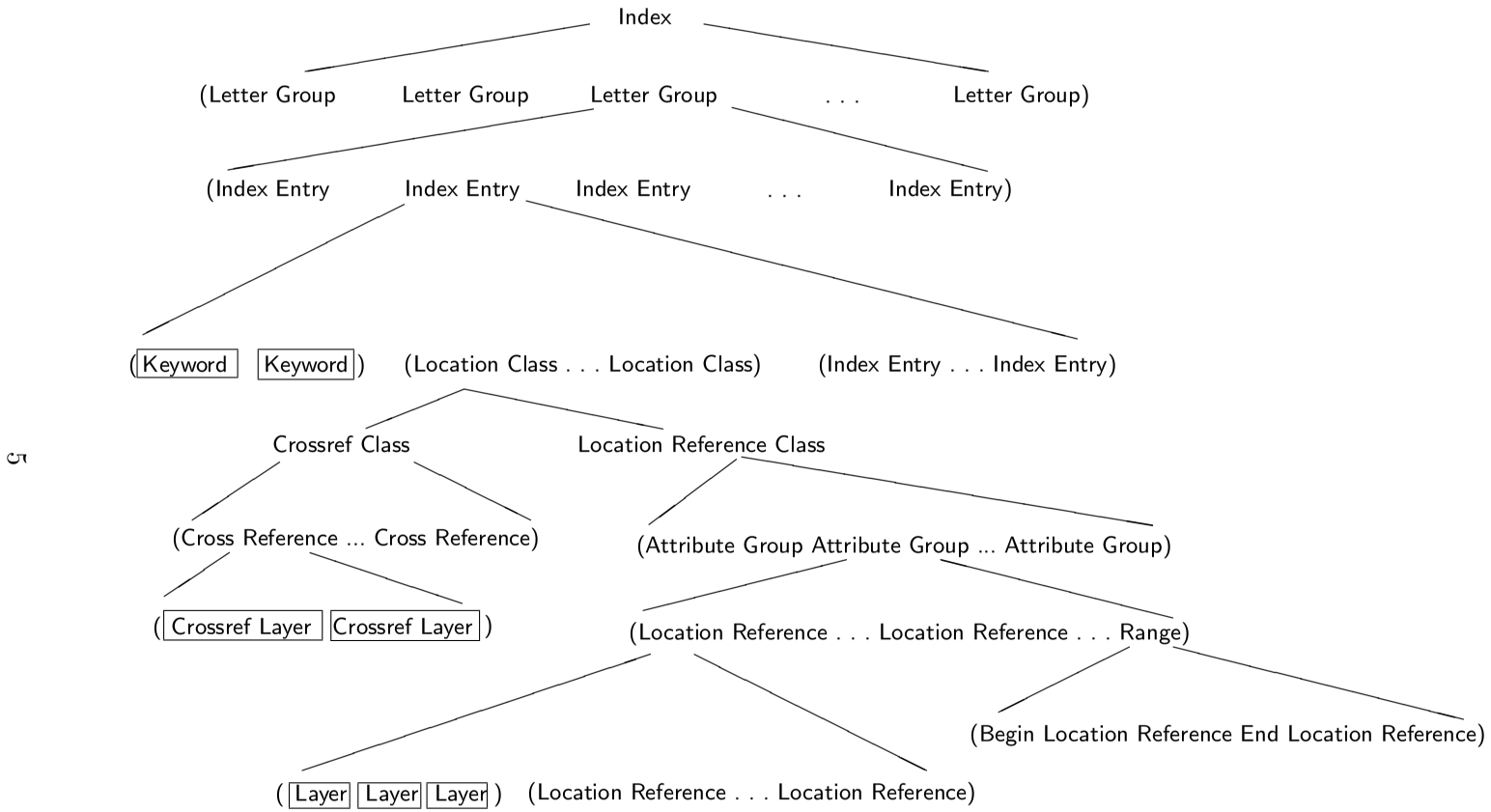

Figure 1: Skeleton tree structure of an index

This method simply outputs the string "`\macro{`" to the markup stream. It is specialised[3] on an attribute instance and a location reference class instance[4]. The above method is specialised on only two of three arguments. Since the argument `depth` is not specialised any further this method matches all location references of the given class and attribute appearing at any depth.

The User Interface

The user interface is primarily responsible to establish appropriate event-bindings. Since event bindings are implemented as CLOS methods, the user interface consists of LISP macros expanding to appropriate method definitions. Defining methods at run-time is a very powerful characteristic of CLOS on which this approach relies heavily.
All markup commands in index style have the following form:

(`markup-`*index-tree-node-name*
[`:open` *markup*] [`:close` *markup*] [`:sep` *markup*][5]
[*context-dependent options such as* `:attr`, `:group`, `:class`, `:depth`, ...])

Since all fields are optional one can assign the markup to only a subset of the context arguments. The accessible context arguments depend on the type of the object that is to be output. We have not made available the whole context to the user, but instead have selected a meaningful subset for each markup command.

The *context-based markup specification* allows, for example, to assign a generic markup for all location reference classes with the command

```
(markup-locref :open  "\generic{"
               :close "}")
```

but redefining the markup for the instances of a certain location class as follows:

```
(markup-locref :class "special-class"
               :open  "\special{"
               :close "}")
```

The event dispatcher has to decide which of the markup schemes is the most specialised one that matches the given arguments. In the above situation the second markup is selected for a location reference of class `special-class`, but for all others the generic markup is used, as defined by the first command.

[3] Specialisation is done on the argument of the `eql`-specialiser. The notation `#<...>` simply denotes an instance of some class.

[4] In the xindy-implementation location classes are themselves instances of another class.

[5] Option `:sep` is applicable for list nodes only.

As an example of the run-time definition of methods, consider the following user interface declaration,

```
(markup-locref :open "\defin{" :close "}"
               :attr  "definition"
               :class "pagenums"
               :depth 1 ))
```

which is specialised on all three arguments, roughly expands to

```
(progn
  (defmethod do-markup-locref-open
      ((attr      (eql #<attribute "definition">))
       (locrefcls (eql #<class     "pagenums">))
       (depth     (eql 1)))
    (do-markup-string "\defin{"))

  (defmethod do-markup-locref-close
      ((attr      (eql #<attribute "definition">))
       (locrefcls (eql #<class     "pagenums">))
       (depth     (eql 1)))
    (do-markup-string "}")))
```

Thus, we use `eql`-specialisation on objects for all three specifiers. Both methods simply output the open (resp. close) tag by calling the function `do-markup-string`.

With this implementation the question of the *metrics* had to be answered that is used to find the *most-specialised* method of a certain set of methods. The problem arises when considering an event with the arguments (*attr=definition*, *locrefcls=pagenums*) that must dispatch to exactly one of the following two methods:

```
(defmethod do-markup-locref-open
   ((attr      (eql #<attribute "definition">))
    (locrefcls class) ... ))

(defmethod do-markup-locref-close
   ((attr      attribute)
    (locrefcls (eql #<class "pagenums">)) ... ))
```

Both methods match the given event but since exactly one method must be called the dispatcher must somehow decide what is the most specialised method of both. Since the method dispatcher of CLOS has a built-in metric that always finds one most-specialised method we simply use it to resolve this kind of ambiguity.[6] If it is

[6] In CLOS the first method would be selected since the first argument *attribute* is more special than the second attribute. We have chosen the positions of the arguments in a way that seems to be most useful in practice.

not resolved as desired, one can define another more-specialised method that resolves this ambiguity explicitly.

Example 1: Tagging of Ranges

A common way of tagging ranges of location references in an index entry is as follows: a range of length 1 is printed with the starting page number and the suffix 'f.', those of length 2 with suffix 'ff.', and all others in the form '*x*–*y*'. Assuming we want to achieve this for the location class *pagenums* we can specify the markup as follows:

```
(markup-range :class "pagenums" :close "f."  :length 1 :ignore-end)
(markup-range :class "pagenums" :close "ff." :length 2 :ignore-end)
(markup-range :class "pagenums" :sep "--")
```

The first two commands specialise on two different range lengths, whereas the third command can be seen as the otherwise-case that is invoked for all ranges which are not of length one or two. The special switch `:ignore-end` causes the second location reference to be suppressed in the resulting output.

Example 2: Tagging of Hierarchies

Sometimes references to appendices are tagged in the following way:

A-1, A-7, A-11, B-3, B-4, B-5, C-1, C-8, C-12, C-13, C-22

This kind of markup can be achieved with the following markup commands:

```
(markup-locref-list       :class "appendices" :sep ", ")
(markup-locref-layer-list :class "appendices" :sep "-")
```

The first command indicates that all location references should be separated by a comma followed by a blank character. The second line says that the different layers the location references consist of, shall be separated by a hyphen character. The location reference *A-1* consists of the following list of layers (*A 1*). Our example could also be output using a much more compact form:

A 1, 7, 11; B 3, 4, 5; C 1, 8, 12, 13, 22

To achieve this markup one needs to transform the location references into a hierarchy. This yields the location reference *A* and its sub-references *1*, *7*, and *11*. The following markup can then be used to obtain the desired result:

```
(markup-locref-list :class "appendices" :depth 0           :sep "; ")
(markup-locref-list :class "appendices" :depth 1 :open " " :sep ", ")
```

The location references at depth zero are separated by a semicolon whereas those at depth one are separated by a comma.

4 Conclusion

We have presented an efficient implementation of a simple context-based markup algorithm in CLOS. Our notion of efficiency mainly concerns the elegance and simplicity of the implementation from the programmers perspective.[7] Our implementation mostly benefits from three interesting properties of CLOS, (a) the ability to define methods at run-time using the macro expansion mechanism, (b) the `eql`-specialisation feature that allows specialisation of methods down to single instances of a class, and (c) the multi-method dispatching paradigm. These features allow to implement the markup algorithm with minimal effort. The resulting code is extremely compact and easily maintainable and extensible. We think that our example is a demonstration of the usefulness of programming concepts such as multi-methods.

5 References

The following books and papers were referenced in this report.

[1] R. P. Gabriel, J. L. White, and D. G. Bobrow. CLOS: Integrating object-oriented and functional programming. *Commun. ACM*, 34(9):29–38, September 1991.

[2] K. Harbo. CoST - Copenhagen SGML Tool. This system is available at `http://www.art.com/cost/`.

[3] S. E. Keene. *Object–Oriented Programming in Common Lisp–A Programmers Guide to CLOS*. Addison-Wesley, 1988.

[4] R. Kehr. x̊indy – Definition of an Indexing Model and its Implementation. Technical Report TI/11, Computer Science Department, Technical University of Darmstadt, May 1997.

[5] M. J. McLennan. The new [incr Tcl]: Objects, mega-widgets, namespaces and more. In U. Association, editor, *Proceedings of the Tcl/Tk Workshop, July 6–8, 1995, Toronto, Ontario, Canada*, pages 151–160. USENIX, July 1995.

[6] J. Schrod. STIL– SGML Transformations in LISP. This system is available at `ftp://ftp.th-darmstadt.de/pub/text/sgml/stil/`.

[7] G. L. Steele Jr. *Common Lisp–The Language*. Digital Press and Prentice-Hall, 12 Crosby Drive, Bedford, MA 01730, USA and Englewood Cliffs, NJ 07632, USA, second edition, 1990.

[7] This does not necessarily mean that the implementation results in a short run-time.