# Adaptive libraries and interactive code generation for Common Lisp

Geoff Wozniak[1] Mark Daley[1,2]
Stephen Watt[1]
[1]Department of Computer Science, [2]Department of Biology
University of Western Ontario, London, Canada
{wozniak, daley, watt}@csd.uwo.ca

**Abstract**

We illustrate the use of a library for an abstract data type whose instances represent the union of various data types and are specialized based on their use. The ADT can be used for a single collection of data that is viewed in different ways in the program. A behavioural analysis determines a specialized type that reflects the use of the data in the program, as well as generating code to define and use the type. The code generation is interactive in that it works in conjunction with a text editor to determine where in the program the specializations are to take place. We present this as a technique for using evaluation to disambiguate code representing many programs and argue it is a useful for design exploration.

## 1 Introduction

In the development of a program, it can be helpful to view one collection of data in multiple ways. In this work, we show a way to represent data as a single entity, then analyze the behaviour of the program in order to create a type for the data. This is realized by using a library for a single abstract data type, called the *Collection dynamic abstract data type* (DADT), that represents the union of multiple abstract data types. An instance of this data type manages multiple structures internally and dispatches operations to the appropriate structures.

The type created by the analysis is made up of a subset of the structures found in the Collection DADT. In this sense, the Collection DADT represents many possible types and its operations can be applied to subsets containing more than one element. For example, the Collection DADT contains both a set and a priority queue, so the `insert` operation may work with any combination of both structures. Constructing the appropriate type is achieved by examining, at runtime, the operations performed on the data and querying the user to disambiguate operations that apply to multiple structures (such as `insert`).

User responses are remembered, in conjunction with their context, to prevent repeated queries.

Once the appropriate type is created (at the user's request), code is generated for type and the operations; however, certain applications of an operation may not operate on all structures within the created type. In the previous example with sets and priority queues, an application of the `insert` operation may have been designated for the set only, whereas another use may be applicable to both. When necessary, a specialized version of the operation is generated for specific call sites. This means that a simple, uniform approach can be used during initial stages of the development to capture the procedure, with refinement coming as a direct result of evaluating the code.

This kind of code generation fits with the call by Smaragdakis [10] for unobtrusive code generation that acts as a tool because annotations in the code are not necessary. Annotations are replaced with interaction, making the directives controlling the creation of code ephemeral. The Collection DADT and its interactive code generation scheme are meant to be used in the context of programming as a design activity, the intent being to write code that represents many possible programs and, through evaluation, narrow the program down to one, unambiguous version.

The next section outlines some definitions and a description of the adaptive library. Section 3 shows a simple workflow involving the library by developing a program to solve a type of word puzzle. Section 4 provides some historical context, motivation and related work, followed by a conclusion.

## 2 An adaptive library

Our analysis approach requires that we be able to produce source code for specific parts of a program and communicate them to the editor. We do this by way of lexical place identifiers. A *lexical place identifier* (LPI) is an identifier for a particular form in the source code. Normally, lexical place identifiers are used to mark forms in the code that are calls to certain functions.

Since LPIs relate directly to the source code as seen by the developer, information about them should be seen in the source code editor. The code generated by our analysis is meant to be later edited by the programmer, thus it does not take place "behind the scenes". In our implementation, LPIs are provided by the editor before evaluation takes place and exist in the dynamic environment of the execution.

The adaptive library is implemented using a *dynamic abstract data type* (DADT) [12]. A DADT is an abstract data type whose instances act as a proxy to different representations that implement some or all of the interface for the data type. Each DADT object has a set of triggers, that is, functions that detect conditions and react them in some manner. DADTs allow for interfaces between multiple abstract data types to be realized as a single ADT whose instances manage their own representation changes.

The adaptive library is for collections, called the Collection DADT. It is meant to facilitate the use of a single collection in different ways. In effect, it is a way to manage independent data structures that contain the same conceptual elements without explicitly creating the data structures in the source code. The interface to the Collection DADT is the union of the interfaces to the different types it manages. Internally, each Collection DADT object maintains a data structure for each type. When an operation is performed on a Collection DADT object, a trigger on the object determines what structures the operation is to be applied to.

When multiple structures are applicable for an operation, there is a potential ambiguity. In the current implementation, the programmer is prompted to choose what structures to operate on. This takes place within the context of a given lexical place identifier and is noted. Any further evaluations within the same LPI are disambiguated based on the programmer's previous answer.

The Collection DADT supports some simple data structures: stacks, associative arrays, queues, priority queues and sets. Some of these structures need parameters, such as how to order the elements (priority queues), how to index the elements (associative arrays) and test for equality (sets). When a collection is created, this information is given via keyword arguments.

# 3 Example workflow

To provide context for the work, we demonstrate the use of the adaptive library through the development of a program to solve a word puzzle. A word puzzle is given by a configuration of cells that hold pieces. Both the cells and the pieces have a specific orientation that does not change. The goal is to place each piece in a cell so that every horizontal (left to right) and vertical (top to bottom) path through the grid forms an English word. See Figure 1 for an example.

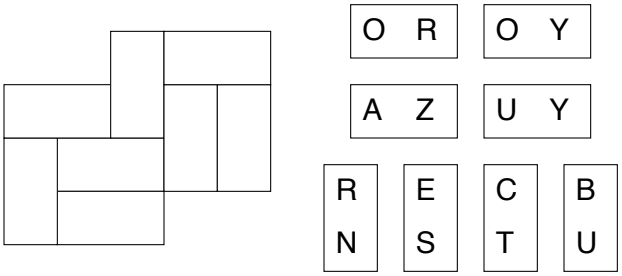


**Figure 1:** *An example of a word puzzle. The solution to the puzzle consists of the words BUY, AZURE, CORNS, TOY making up the horizontal paths, and the words ACT, ZOO, BURY, URN and YES making up the vertical paths. This particular instance is © 2008, Fraser Simpson.*

A puzzle is given as a set of cell descriptions, each cell being either horizontal or vertical with a unique identifier. Another set provides the connections between cells using their identifiers.

A simple search will be used to solve the puzzle: Choose an empty cell and find a piece that fits in the cell that does not violate the constraints, that is, if the piece would complete a word, the word must be legal. If no such piece can be found, backtrack to the last cell filled. Continue until either all the pieces are placed or no valid configuration can be found.

To implement the algorithm, we look at the collection of cells as three types: an associative array, a set and a priority queue. The associative array is used to index cells by their identifiers for lookup during the creation of a puzzle. When a connection is processed, the cells named in the connection are retrieved and the appropriate connection is made. The set is for iterating over the cells, used when printing the solution and for debugging. The priority queue is used when choosing the next cell. We will use the Collection DADT to represent the cells, preventing the need to manage three structures holding the same data.

To make the collection, we use `make-collection` and provide it with information on how to prioritize the cells and how to index them. The code for creating the collection of cells is

```
(make-collection :prioritize-by #'cell-compare :index-by #'id).
```

We must also test for equality, but this done with `eql` by default and is sufficient. This form is marked in the text editor with a lexical place identifier, but does not change the source code. Instead, the LPI is inserted when the code is evaluated, which must be initiated in the editor. This permits any objects created by this form to be tagged with the LPI so that the code suggested by the analyzer can be communicated to the editor.

To create the puzzle, we first populate the set and associative array so that the connections can be processed. This is done using the `insert` operation. However, the `insert` operation is applicable to sets, associative arrays and priority queues. This ambiguity is resolved by querying the user to indicate which structures the operation is applicable to and is done when the code is evaluated.

Once the connections are processed, the priority queue is populated, also with calls to `insert`, and again the user is queried to indicate the structure to operate on is the priority queue.

The function implementing the main algorithm for solving the puzzle is given in Figure 2. The highlighted forms are those that operate on the collection of cells and are associated with lexical place identifiers. All of the operations indicated by LPIs are meant to be applied to the priority queue. In the case of `insert`, its initial use must be disambiguated.

Note that the `insert` operation is used uniformly throughout the program: it is always applied to the collection of cells with no syntactic indication of what internal structures the operation is applied to. This permits parallel processing of structures without explicitly writing code to do so. Furthermore, such uniformity permits changing the structures processed by changing interface operations. In Figure 2, for example, we could change the algorithm to

```
(defun solve-puzzle (puzzle pieces)
  (let ((next-cell (unless (empty-p (cells puzzle))
                     (extract-min (cells puzzle)))))
    (unless next-cell (return-from solve-puzzle puzzle))
    (loop for piece in pieces
          if (and (place-piece next-cell piece)
                  (configuration-legal-p puzzle)
                  (solve-puzzle puzzle (remove piece pieces)))
            do (return-from solve-puzzle puzzle)
          else do (withdraw-piece next-cell))
    (insert (cells puzzle) next-cell)
    nil))
```

**Figure 2:** *The main algorithm implementing the logic to solve the puzzle. The forms given lexical place identifiers are highlighted.*

use a set instead of a priority queue by changing the `extract-min` operation to `select-element` and re-marking the form to indicate it has changed[1].

## 3.1 Analysis and specialization

After the code has been evaluated and we are satisfied with the correctness insofar as it solves the problem at hand, we perform an analysis. The analysis determines what structures are to be used at the lexical places and how they are to be represented.

First, the analysis looks at what structures were used at each lexical place. Only the structures used are to be included. This means that in order for the analysis to capture the intent of the programmer with respect to what structures are to be used, there must be sufficient code coverage through testing.

Objects are grouped by their tags, the tags being the lexical place identifier of the form that created them. These tags are known as *creation places*. An object tagged with $p$ is written as $O_p$. Objects tagged with $p$ may be used at another place $q$. The analysis phase determines the set $Q_p$, where $q \in Q_p$ if an interface operation was performed at lexical place $q$ on a structure within an object $O_p$. For each $q \in Q_p$, the set of structures within $O_p$ accessed at $q$ is denoted $s(q)$. The set of structures required for objects created at $p$ is then $T(p) = \bigcup_{q \in Q_p} s(q)$. A type $T_p$ is created to hold the set of structures found in $T(p)$ for each creation place $p$. The type is realized as a class with a slot for each structure.

An analysis is performed when a specialization is requested by the user in the editor after selecting a form corresponding to a creation place. For a creation place $p$, the specialization will present code that creates an object of type $T_p$,

[1] Of course, this might require changes in other parts of the program, such as making two copies of a set in the collection of cells.

using the options provided. `defclass` and `defmethod` forms are generated for objects of type $T_p$ and the bodies of the methods perform the operation on the necessary structures. Using methods in this fashion means that the lexical places denoting calls to the interface functions do not have to be changed.

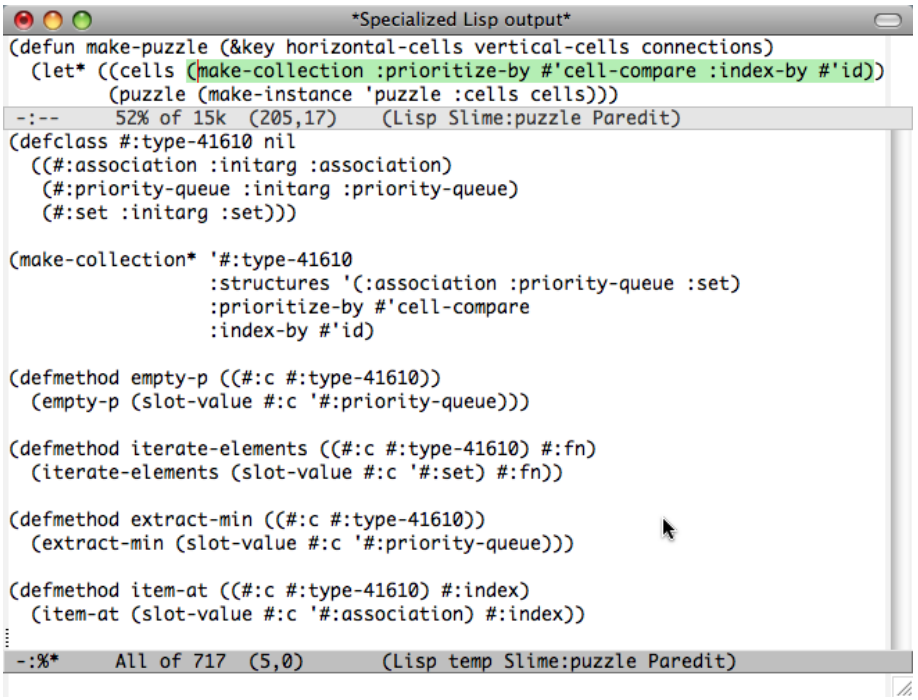


**Figure 3:** *A screenshot of a specialization request. The original source code is in the top (narrow) pane, with the generated code in the bottom pane. The generated call to make-collection* is to replace the highlighted call in the source code.*

Note that the specialization has to be requested and that the code is not immediately inserted into the source file. We have implemented the interaction required for specialization using Emacs and SLIME[2] and modeled the interface after its version of macroexpansion. That is, a separate window is presented in conjunction with the source code window in order to see what the creation place would be replaced with, as well as the class definition and method forms. Within the expansion window, you can initiate a replacement of the form at the creation point and insert the other forms into the source code. Figure 3 contains a screenshot of a specialization request with some of the generated code.

The generated code shown in Figure 3 does not include a method definition for `insert` because it is not uniformly applicable to the type. Instead, the specific call sites have to be specialized. For example, the call to `insert` in Figure 2

[2]The Superior Lisp Interaction Mode for Emacs. See `http://common-lisp.net/project/slime/`.

gets translated into

```
(dapply (#:priority-queue) insert (cells puzzle) cell)
```

which is a macro that properly evaluates the arguments and calls the operation on the slots indicated (in this case, the slot holding the priority queue). The code replacement for each such call site is provided from within the specialization request and allows the user to replace code one form at a time, similar to search and replace behaviour found in many editors.

# 4 Discussion

The idea that programming is a design activity as expressed by Reeves [9] can be traced back to Naur's notion that programming is theory building [6]. Naur argues that the programmer accumulates knowledge of what the problem is and how to solve it by experimentation without any particular method. This is known as the Theory Building View of programming. The Theory Building View espouses the notion that there are many tools and techniques available to the programmer in order to determine the structure of the program, and that there is no specific order in which these elements need be applied in order to determine the structure.

The work described in this paper starts from these assumptions. The question we asked is, what kinds of programming environment elements might be useful when the program code is considered to be the design document? That is, what might be useful in exploring the design space?

Looking at agile programming as described by Cockburn [1], the act of development involves communication between those involved in the development. At the risk of anthropomorphizing the development environment, we note that the development environment has ready access to the history and execution information of the program, so it seemed pertinent to harness that information. Combining access to the information with some simple inference rules means the system can, at the very least, make suggestions with respect to source code augmentations. The results need not be correct in the sense that it validates the code if considerable ambiguity is involved, but the results should strive to be relevant.

Osterweil argues that process descriptions grow out of processes, that is, the act of working through a process is an excellent way to come up with a process description [7]. He further argues that software processes are themselves software and thus, can be described as a kind of program. He stresses, however, that the high-level descriptions in such a program may be very difficult to specify precisely [8]. Filling in the details may be garnered from observing processes and although the details may not be universally applicable, they do address the tasks at hand. From this observation, we aimed at facilitating dynamism in the design process and using behaviour to infer something about structure.

That said, the interactive nature of the work suggested that the application of any results should not take place automatically in the sense of "Do what I

mean" (DWIM). The DWIM approach can be seen in some languages, such as Perl and PHP, where an expression such as `"10 elements" + 4` might evaluate to 14.[3] This can be particularly insidious when variables are used and they are bound non-locally. A more earnest attempt at DWIM was made by the Interlisp system [11]. However, well-meaning changes still manifested at run-time, such as interpreting an atom as a single element list when the procedure expected a list. We felt that applying the results of our analysis to the code or the execution of the code without confirmation would be misguided, since it would increase the complexity of evaluation and potentially make debugging more difficult. Instead of seeing it as "Do what I mean", we took the approach of "Is this what you mean?"

Building data types through interaction to pare down a very general representation is an approach to program design in the spirit of Kiczales' notion of open implementations [3]. The Collection DADT acts a meta-interface to a collection of data structures and the code is the corresponding meta-program that is eventually specialized. The code generated is meant to replace the meta-program, however, it may be worthwhile to consider using it in the form of a presentation extension as demonstrated by Eisenberg [2].

The analysis in the work presented is reminiscent of attempts at automatic data structure selection as described by Low [4, 5]. Low's techniques applied to a single abstract data type with multiple representations. The Collection DADT is different in that it is more aptly described as a union type. The analysis and specialization phases described above essentially build the type based on observed behaviour. The type may not be exactly what is represented by the Collection DADT. In a sense we do select a data structure, but it is combined from different ADTs and not different representations of the same ADT.

# 5 Summary and conclusion

We have shown an approach to interactive code generation by specializing data types. The data types are used within the program as instances of an abstract data type representing the union of multiple abstract data types. The library providing the general ADT collects profiling information about where in the code its instances are used based on information from the source code editor and the programmer. These places can then be specialized to the types observed to be used during execution.

The purpose of this approach is to explore the notion that programming is a design activity and provide tools for exploring the design space. In this sense, we support incomplete and ambiguous programs, eventually working toward a more detailed specification based on behavioural aspects.

[3] Indeed, this is the case for both Perl and PHP.

## Acknowledgements

We thank the anonymous reviewers of the paper for constructive criticism and helpful suggestions in order to improve the work.

## References

[1] A. Cockburn. *Agile Software Development: The cooperative game*. Agile Software Development Series. Addison-Wesley, 2nd edition, 2007.

[2] A. D. Eisenberg and G. Kiczales. Expressive programs through presentation extension. In *AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development*, pages 73–84, New York, NY, USA, 2007. ACM.

[3] G. Kiczales. Beyond the black box: open implementation. *Software, IEEE*, 13(1):8, 10–11, Jan 1996.

[4] J. Low and P. Rovner. Techniques for the automatic selection of data structures. In *POPL '76: Proceedings of the 3rd ACM SIGACT-SIGPLAN symposium on Principles on programming languages*, pages 58–67, New York, NY, USA, 1976. ACM Press.

[5] J. R. Low. Automatic data structure selection: an example and overview. *Communications of the ACM*, 21(5):376–385, 1978.

[6] P. Naur. *Computing: A Human Activity*, chapter Programming as Theory Building, pages 37–48. ACM Press, 1992.

[7] L. Osterweil. Software processes are software too. In *ICSE '87: Proceedings of the 9th international conference on Software Engineering*, pages 2–13, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.

[8] L. J. Osterweil. Software processes are software too, revisited. In *ICSE '97: Proceedings of the 19th international conference on Software engineering*, pages 540–548, New York, NY, USA, 1997. ACM.

[9] J. W. Reeves. Code as design. *developer.* Magazine*, 2005. (Originally appeared in the *C++ Journal*, Fall 1992.)

[10] Y. Smaragdakis. *Domain-Specific Program Generation*, chapter A Personal Outlook on Generator Research. Lecture Notes in Computer Science. Springer-Verlag, 2004.

[11] W. Teitelman and L. Masinter. The Interlisp Programming Environment. *Computer*, 14(4):25–33, 1981.

[12] G. Wozniak, M. Daley, and S. Watt. Dynamic ADTs: a "don't ask, don't tell" approach to data abstraction. In *International Lisp Conference*, pages 209–220. The Association of Lisp Users, April 2007.