UNIVERSITY OF BUEA

Faculty of Science

Department of Computer Science

BACHELOR OF SCIENCE IN COMPUTER SCIENCE

Project Report

# Towards a Data Structure for Parameter Passing Styles

NOUMBA LEONARD

SC17A350

SUPERVISOR: William S. Shu, PhD

SEPTEMBER 2020

# DECLARATION

I hereby declare that this project report has been written by me Noumba Leonard, that to the best of my knowledge, all borrowed ideas and materials have been duly acknowledged, and that it has not receive any previous academic credit at this or any other institution.

Noumba Leonard
SC17A350
Department of Computer Science
Faculty of Science, University of Buea

# CERTIFICATION

This is to certify that this report entitled TOWARDS A DATA STRUCTURE FOR PARAMETER PASSING STYLES is the original work of NOUMBA LEONARD with Registration Number SC17A350, student of the Department of Computer Science at the University of Buea. All borrowed ideas and materials have been duly acknowledged by means of references and citations. The report was supervised in accordance with the procedures laid down by the University of Buea. It has been read and approved by:

| | |
|---|---|
| William S. Shu, PhD CEng CITP MBCS<br>University of Buea<br>(Project Supervisor) | Date |
| Dr. Denis L. Nkweteyim<br>Head of Department of Computer Science | Date |

# DEDICATION

I dedicate this work:

To the memory of my Uncle Numba Emmanuel who was a father to me. Thanks so much for your love and care that still give me inspiration to date.

To God almighty for his endless strength and blessings that have carr~~y~~ied me through this project.

To my mother for her everyday love, care and support.

# ACKNOWLEDGEMENTS

I take this opportunity to express my profound gratitude and deep regards to my guide (project supervisor) William S. Shu, PhD for his guidance, monitoring and constant encouragement throughout the course of this project.

I would like to thank the following people who helped make it possible for this project to be completed. First of all, thanks to my mother, whose advice and experience added to a more concise focus in the work as a whole. Also, I want to thank my course mates, Chiatiah Carlson, Mark Ngoran, Suh Edmond, Claude Nkeng, and Layu Romaric who provided me with their scholarly skills, candid opinions and endless optimism.

Finally, I would like to thank God, for letting me through all the difficulties. I have experienced your guidance day by day. I will keep on trusting you for my future.

# ABSTRACT

Parameter passing styles are the various ways used to pass parameters to procedures or functions. A parameter passing style hugely depends on how we intend to use the parameter and what factors characterise it. Several factors such as context, evaluation, and typing have been exploited and used to describe how parameters are passed. Parameter passing styles modify and significantly affect the meaning of computations and so~~, parameter passing styles~~ they are widely exploited in programming languages. In this project, we construct a structure called Passing Style Container (a box for holding parameter passing styles) and define permissible operations on the passing style container. That is, a passing style container (PSC) for major known passing styles and possibly infinitely many user define styles added to the PSC at runtime. The PSC allows users add new passing styles, remove undesired or unpleasant ~~stylo~~es, as well as add/remove interpretations of the added/removed passing styles. We then illustrate usefulness of one of these passing styles in safety systems. Specifically, parameter passing by value is used to prevent any changes to an entity before or after the entity is used.

TABLE OF CONTENTS

# Chapter 1
# Introduction

## 1.1 Project Motivation

Rapid growth of programming languages and software systems has increased the need for efficient and more reliable passing styles for communication between modules (or functions) of programming languages or systems and also across different application domains. With the ever-increasing data to be communicated between different application domains, there is the need for an efficient structure to hold the data and a passing style to safely communicate the data in an efficient way.

We therefore seek to develop a Passing Style Container (PSC), which explores the various factors known to affect parameter passing styles. Also, we identify values of factors (known to affect parameter passing styles) that are predicted to yield good performance. In addition, we develop functions (operations) for manipulating the PSC. Some of the permissible operations on the PSC include the following:

- Add a new parameter passing style.
- Remove an existing passing style.
- See various passing styles in the PSC.

My interest in this project is to model and develop a passing style container (PSC) that can serve users to communicate and protect their data between application domains. Also, due to the ever-increasing amount of data to communicate, users can define and add new, even more efficient passing styles in the PSC. The ability of the PSC could achieve the evolving need for efficient parameter passing.

## 1.2 Project Aim and Objectives

The aim of this project is to develop a PSC that closely examines how to pass parameters

(including novel styles) and takes into account factors that affect parameter passing such as entity passed (e.g. value or computation), evaluation, execution context and typing.

The objectives of this project are to:

- Identify and group the various parameter passing styles.
- Develop a passing style container (PSC) that holds parameter passing styles.
- Identify the various parameter passing styles with specific application domains and also explore usefulness of novel styles.

~~**Specific problem/solution instance**~~

In this project we develop a passing style container which contains parameter passing styles. A list data structure is used to model the passing style container which holds a collection of passing styles. This data structure is used because it is dynamic, immutable and one can add or remove from it easily.

A record data structure is used to model the various parameter passing styles. This data structure is used because records can hold values of arbitrary types.

### 1.3 Report Structure

The rest of this report is organized as follows. Chapter 2 explores the analysis and design of the data structure wherein we define the problem statement, the research aims and questions and finally the design algorithms of the program. Chapter 3 presents the implementation part of the project. Chapter 4 presents the results of the project and discusses the work. The chapter provides results of implementation and explains the algorithm used to implement the main activities. Chapter 5 is the conclusion of the report.

# Chapter 2

# Analysis and Design

## 2.1. Requirements of the System

The PSC provides users with the ability to add new passing styles, remove an existing passing style and also see all available passing styles. User-defined passing styles can then be used by other organisations or users if need be. In order to achieve this, we ensure that:

- Users can get hold of the available factors for parameter passing.
- Users can create their own passing style using any number of the available factors affecting parameter passing.
- Users' newly created passing styles are added to the PSC~~structure~~.
- Users can remove a passing style they don't desire.
- Users are able to see list of available passing styles.

## 2.2. Main Entities, Activities and Data Structures

Parameter passing styles are assorted and widely used in programming. The choice of a parameter passing style is important in the design of a high level programming language. The main entities refer~~s~~ to the factors exploited and used to describe how to pass parameters. They ~~main entities~~ include:

a. **Entity passed**: Entity passed refers to the parameter that is passed to a method and used to communicate data between modules (or functions). The types of entities passed include:
   - ***Value***: A value is a configuration of states or final result that cannot be simplified further and considered ok.

- ***Reference***: A reference is a memory location or address of a value.
- ***Computation***: A computation is a possibly non-terminating expression that could be further simplified.
- ***Denotation***: Denotation is the meaning attributed to an expression or object from a mathematical set called a domain.
- ***Continuation***: A continuation is the rest of a computation, after just after the point where a given expression has been evaluated so far.
- ***Environment***: An environment refers to the context in which the association of values with variables are found. It maps variables to semantic values (constants or closure) [1].
- ***Object***: An object refers to a collection of data together with functions to operate on that data [3].

**b. Context**: Context refers to the environment where a function or parameter is called and evaluated. Context describes occurrences inside programs where reduction may actually occur [1]. This can be in the body of the called or the caller procedure.

**c. Evaluation strategy**: An evaluation strategy is a set of rules for evaluating the arguments of a function call and what kind of values to pass to the function. It defines the order in which redexes must be reduced. The eager and lazy Eevaluation strategies are the two most common ones and usually used to classify functional programming languages. We shall assume at least these two hereied as eager or lazy. In eager evaluation strategy, the arguments of a function are completely evaluated before the function is applied. In lazy evaluation, arguments to a function are evaluated only where their values are needed in the evaluation of the function body.

**d. Typing:** A type refers to a collection of values that share some property. Typing generally refers to use of types in order to both capture invariants (a property that is expected to always hold) of programs (or functions), and to ensure its correctness and hence definitely exclude certain classes of programming errors. Typing also help maps variables (or constants) to types.

### 2.3. Description of passing styles

The combinations of ~~the above~~ factors that affect parameter passing as stated in section 2.2 above~~, their types, and values, affects the computation carried out~~. An instance of the combination of the above factors and its use is broadly considered as a passing style. This is illustrated below for known parameter passing styles:

- **Call by value:** In call by value, the entity passed is a value and the argument is evaluated in the context of the caller procedure at the time of procedure call. The entity passed is evaluated in order (typically from left to right). Any changes to the value inside the called procedure is purely local to the called procedure~~,~~ and, therefore, not visible inside the caller procedure [[2]].

- **Call by reference:** Here, the entity passed is a reference (an address in memory) and the argument is evaluated at the time of procedure call in the context of the caller procedure. The reference passed as an argument can be modified inside the called procedure with visible effects inside the caller, after the call [[2]].

- **Call by copy-restore:** Call by copy-restore is similar to call by reference. The argument passed is not modified. Call by copy restore avoids this modification by leaving the result of the evaluated argument in the caller's environment.

- **Call by name:** Call by name is similar to call by value. However, t~~T~~he ~~values of~~ arguments are not evaluated at the time of procedure call. The arguments are substituted into the function body and evaluated only when used and as many times as they are used. A function is created for each argument and each time the argument is needed, the function is called which evaluates and returns the argument.

- **Call by need:** Call by need is similar to call by name. When an argument is evaluated the very first time, its value is used in all occurrences of that argument, thus avoiding its re-evaluation.

- **Call by sharing:** In call by sharing, the entity passed is an object. The argument objects

are shared between the caller and the called procedure. If the called procedure modifies a shared object, the modification is visible to the caller procedure on return.

### 2.4. Activities

Activities refers to the possible operations that can be performed on the Passing Style Container (PSC) and ~~also on its elements which ar~~ the parameter passing styles it contains~~n this case~~. The main activities (operations) here include:

- Add new passing styles: New passing styles are constructed by selecting a combination of parameter passing factors and a name. A function is used to add the style to the PSC.
- Remove passing styles: Removing an existing passing style from the PSC was achieved by using a function that takes parameter passing style name and remove the style from the PSC if it exists.

### 2.5. Design

This project was developed using the prototyping/exploratory programming approach to problem solving ~~wherein~~and in it the development process was broken down into smaller task to ease the management of the data for passing styles within the data structure. This equally made the developed data structure for parameter passing appropriate for testing.

#### a) Main structures

To model parameter passing styles, a record data structure is used to hold parameter passing styles which is made up of a passing style name and factors as shown in Figure 2.1:

```
passing_style is record    /*declaring passing style as record*/
        name: string;       /* record fields (name, factor) */
        factor: factors     /* factors denotes the type of the field factor */
end record                  /*end of passing style record*/
```

Figure 2.1: Passing style structure.

A record data structure [Figure 2.1] is used to hold a passing style. The record fields (name and factors) represents the passing style name and factors respectively.

```
type factors is record                    /* declaring factors type as record*/
        entity : entity_passed;    context: context_type;
        evaluation : evaluation_strat; typing : typing
 /* the fields (entity,context etc.) denotes the factors known to affect parameter passing*/

end record  /*end of factors record*/
```

Figure 2.2: Passing style factors.

A record data structure [Figure 2.2] is used to hold factors for parameter passing styles. The record fields are the known factors assumed to affect parameter passing.

```
type entity_type = Entity_type of factor_instance_list  /*various entity types*/
type context_type = Context_type of factor_instance_list /*various context types*/
type evaluation_strat = Evaluation_strat of factor_instance_list   /*various evaluations strat*/
type typing =Correct_type of factor_instance_list
type factor_instance_list = int list /* list for holding various instances for a factor*/
```

Fig.2.3 : Factor types

Figure 2.3 shows the factors known to affect parameter passing. A union type of list (entity_type for entity) [Fig.2.3 ] is used to hold together all possible values for a factor.

A list data structure [Fig.2.4] is used to hold the collection of parameter passing styles including novel styles so as to capture the relationship among them.

```
PSC is list     /*declaring the passing style container as list*/
style1 : passing_style; style2 : passing_style;  style3 : passing_style  /* passing styles*/
end list        /*end of PSC list*/
```

Fig.2.4: Passing style container.

Figure 2.4 shows the PSC for parameter passing. Its elements are parameter passing styles [Figure 2.1]. Permissible operations are used to perform the intended actions (add new passing styles, remove passing styles, see all passing styles) on the PSC.

An association list data structure [Figure 2.5] was used to hold interpretations for the various passing styles. The association list associates a given interpretation with a passing style name as shown in [Figure 2.5]:

```
Interpretation_structure is list   /*declare the structure for holding interpretation as list*/
/* interpretation for the passing styles style1 and style2*/
        style1_interpretation: (style_name: string , interpretation_text: string)
        style2_interpretation: (style_name: string , interpretation_text: string)
/*style_name represents the name of the style and interpretation text refers to the
interpretation (in prose) of the passing style*/
End of interpretation structure
```

Figure 2.5: Interpretation structure

New passing styles interpretation are added to the interpretation structure [Figure 2.5] (at runtime) immediately after the style is added to the PSC. An ~~function (~~operation~~)~~ that takes a passing style name is used to retrieve an interpretation for a passing style.

**b) ~~Design~~Main algorithm**

```
Start
 REPEAT
(1) Display menu    /*operation that can be performed*/
(3) Read selected_operation      /*operation entered by the user*/
(4) Switch (selected_operation)
      Case (Add passing style):     /*add new passing style operation*/
      PSC <-- Add_passing_style _PSC Style
       Break;
     Case (Remove passing style):
     Passing_style_list <-- Remove_passing_style PSC Style
      Break
     Case (See all passing styles):
     Display PSC   /*display the passing style container*/
       Break
     Case (Select style from structure):
     Display_all_info_about_selected_style  Style
      Break
     Case (Exit):
          Exit = true
      Break
UNTILL Exit = true
Stop
```

Fig. 2.6: Design algorithm

Fig 2.6 shows how the main algorithm ~~on how the system~~ for ~~developing~~ the passing style container works. ~~There are a number of actions that could be performed on the system.~~ As the system starts, an action menu is displayed and one gets to select an action (operation). Some of the action menu items include:

- Add passing style and interpretation.
- Remove passing style and interpretation.

Depending on the user's choice, a given action is carried out in the system. Except for the exit menu item (not listed above) which exits the app. Every other menu item performs the action under it and gives the user the opportunity of performing another. [Figure 2.7] ~~and [Appendix 1]~~ reveals some critical tasks carried out in the system, notably~~,~~ the adding of new passing styles ~~and r~~Removing of passing styles is given in Appendix 1~~respectively~~.

```
(1) Start
(2) entity = [a], context = [a], evaluation = [a], typing = [a]  /* 'a' = initial value for factors*/
REPEAT
(3) Display factors   /*factors known to effect parameter passing*/
(4) Read selected factor   /*factor entered by the user*/
(5) Read Number N          /*number N to initialize selected factor*/
(6) Switch (selected_factor)
    Case (entity):
      entity <-- add_to_list entity N /*adding N to list that holds values for selected factor
(entity in this case)*/
         Break
      Case (evaluation):
      evaluation <--  add_to_list  evaluation  N
         Break
      Case (context):
      context <--  add_to_list context N
         Break
      Case (typing):
      typing <--   add_to_list typing  N
         Break
(7) IF (more factors?) then exit = false else exit = true
    UNTILL exit = true
(8) Read name                                   /*name for the passing style*/
(9) factors <-- preserve_default {entity ; context; evaluation; typing}    /*preserve default
values for factors not of interest to the user but removes default values for selected factors*/
(10) Passing_style <-- {name; factors}                    /*creating the passing style*/
(11) PSC <-- insert PSC Passing_style    (*inserting the new style to the PSC)
(12)End
```

Fig. 2.7 Algorithm to add new passing style.

[Figure 2.7] shows the algorithm for creating and inserting a passing style into the PSC.

# Chapter 3

## Implementation

We implement the sets of permissible operations used to carry out operations on the developed structure. All programs and code fragments are implemented in the OCaml programming language version 4.10.0, on a 64 bit computer with an MS Windows 10 operating system.

### 3.1. Implementation of parameter passing styles

The implementation of parameter passing styles designed in [section 2.5], [Figure 2.1], [Figure 2.2], and [Figure 2.3] are all implemented in [Figure 3.1].

```
type  passing_style = {name: string;    factor: factors }        /*parameter passing style */
type factors = {entity : entity_passed;    context: context_type;    evaluation : evaluation_strat;
typing : typing }   /*parameter passing style factors*/
type entity_type = Entity_type of factor_instance_list      /*various entity types*/
type context_type = Context_type of factor_instance_list
type evaluation_strat = Evaluation_strat of factor_instance_list
type typing =Correct_type of factor_instance_list
type factor_instance_list = int list   /*list that holds instances for factors*/
```

Figure 3.1: Implementation of passing style components.

[Figure 3.2] shows the function used to construct parameter passing styles from a suitable combination of factors and passing style name. The system works internally with numbers.

```
/*function (sanitize) that constructs a passing style */
/* user_styles, name: list of known passing styles and name of passing styles respectively*/
/* entity_type, context_type, evaluation_strat, typing: factors that effect parameter passing*/
  let sanitize entity_type context_type evaluation_strat typing user_styles name=
    match (entity_type, context_type, evaluation_strat, typing) with
    (Entity_type [1], Context_type [1], Evaluation_strat [1], Correct_type [1]) -> user_styles
    | (Entity_type [1], Context_type [2], Evaluation_strat [3], Correct_type [4]) -> user_styles
    | (Entity_type _, Context_type _, Evaluation_strat _, Correct_type _) -> let new_style =
  let x = {entity = entity_type; context = context_type; evaluation = evaluation_strat; typing =
typing}           /* record that holds passing style factors with their values */
  in let y = {name = (setName name); factor = x} /*constructed passing style*/
  in let () = print_string "\npassing style " in let () = print_string """ in
  let () = print_string name in let () = print_string """
  in  insert_new_style user_styles y
  in new_style
```

Fig 3.2: Function for creating passing styles

The record (y) used to hold the parameter passing factors (x) and the name (name) as indicated in [Figure 3.2] defines a parameter passing style. This style (y) is then added into the list of styles by the function insert_new_style.

```
{name = Passing_style "Pass by XXXXX";   /*passing style name*/
   factor = {entity = Entity_type [1; 2]; context = Context_type [3];
          evaluation = Evaluation_strat [1]; typing = Correct_type [1]}
  /*passing style factors alongside their  values*/             }
```

Fig. 3.3 Newly created style

Figure 3.3 shows a newly created style. The system works internally with numbers.

### 3.2. Implementation of main structure for passing styles

As mentioned earlier, a list data structure was used to implement the passing style container (PSC). The PSC holds the collection of passing styles which are constructed as shown in [Figure 3.2]. The implementation of the PSC for parameter passing styles ~~designed in [section 2.5], [Figure 2.4]~~ is implemented in ~~[~~Figure 3.4~~]~~.

```
let user_styles = [
{name = Passing_style "Pass by Value";
factor = {entity = Entity_type [1]; context = Context_type [1];
          evaluation = Evaluation_strat [1];  typing = Correct_type [1]}};
 {name = Passing_style "Pass by Reference";
   factor = {entity = Entity_type [2]; context = Context_type [1];
          evaluation = Evaluation_strat [3]; typing = Correct_type [3]}}
         ]
```

Fig. 3.4 Implementation of PSC

### 3.3. Operations on main structure

The basic operations performed on the PSC include: Adding new passing styles, removing an existing passing style and see all available styles in the PSC.

#### a. Add new passing styles

```
 /*function to insert passing styles into the passing style structure*/
let rec insert_new_style user_styles y =  /*y refers to the style to be inserted*/
    match user_styles with                    /*user_styles refers to the passing style container*/
    [] -> let () = print_string "successfully added.\n\n" in  [y]
   |h :: t ->if h = y then let () = print_string " already exists.\n"
 in  user_styles else h :: insert_new_style t y
```

Fig. 3.5: Add new style to PSC

[Figure 3.5] shows the function used to insert newly constructed passing styles into the PSC. This function is being used in the “function for creating passing styles” [Figure 3.2] to add new passing styles in the PSC immediately after they are created. Parameter passing style factors are constructed by selecting factors of interest alongside their values from each class of factors. Implementation of the function to remove passing styles from the structure is given in [Appendix 3].

# Chapter 4

## Results and Discussions

In this section, some of the results obtained after implementing all the main operations is presented. This program makes use of a Command Line Interface (CLI) to serve as a medium of communication between the user and the application. This means that the system will only recognize text-like commands when a user is “talking” to it.

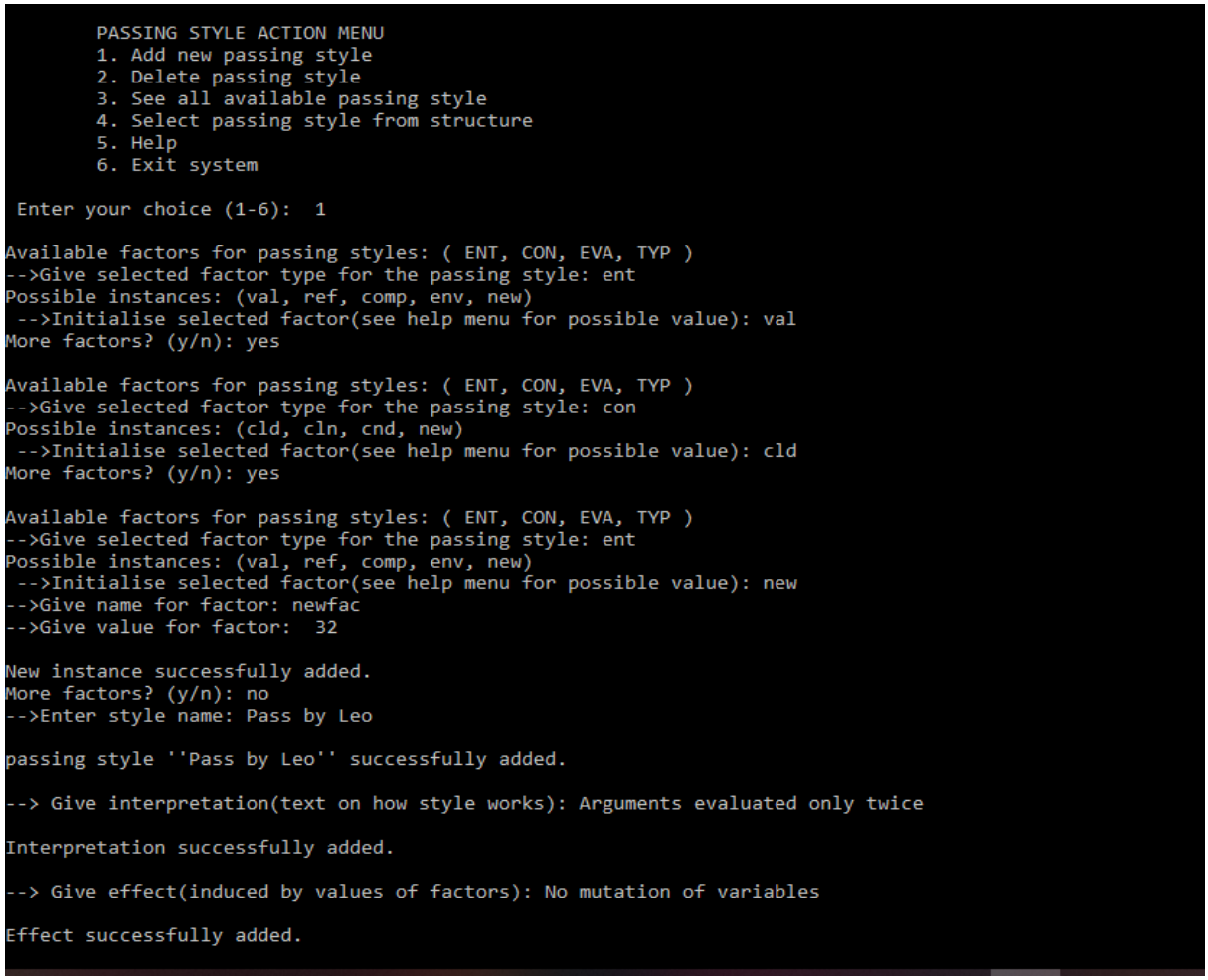


Fig. 4.1 Add new passing style

Figure 4.1 demonstrates how new parameter passing styles are created and added to the PSC alongside its interpretation and effects. One selects a factor of interest from a list displayed, and a list of known values for that factor is also displayed for selection. A new instance for a factor is created by entering the string “new” for the factor’s value when initialising it. See, the third available factor entered in [Figure 4.1].

Once a factor is selected and initialised, the user can decide to add more factors of interest. Users can provide more than one instances for a factor by selecting the factor more than once. A list of known values for the selected factor is displayed for selection each time the factor is selected as shown in [Figure 4.1]. Default values for factors are assumed if the user does not provide them.

After initialising a selected factor, one gets the option to select more factors of interest. When user selects no more factor, the passing style name is entered. The passing style is then created with values for the selected factors of interest and inputted name. The newly created style is then added to the PSC and it’s indicated by a success message.

After adding the passing style to the PSC (indicated by the success message), the interpretation and effect of the newly created passing style are entered as shown in [Figure 4.1].

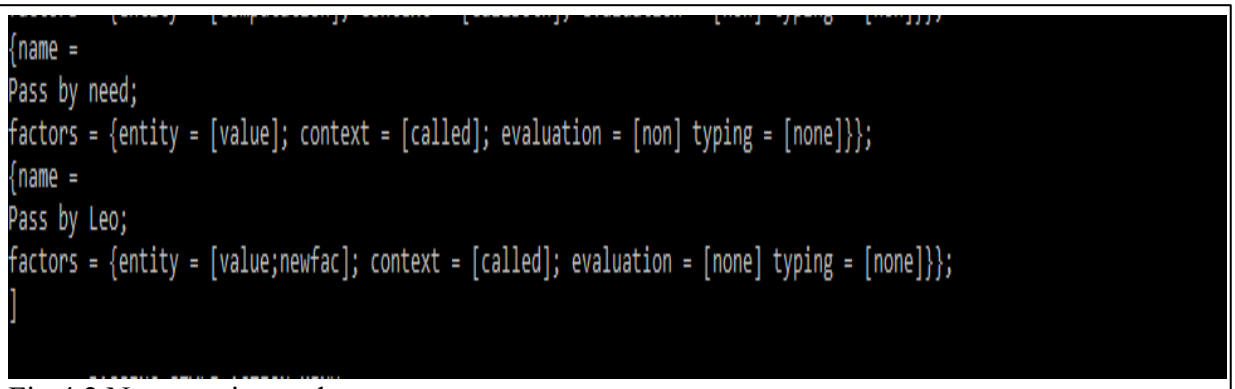

Fig.4.2 New passing style

Figure 4.2 shows the presence of the newly added style in the PSC alongside passing styles known to the system. It also shows the newly defined instance “newfac” for the factor “entity”.

See, second passing style (“Pass by Leo”) displayed in [Figure 4.2].

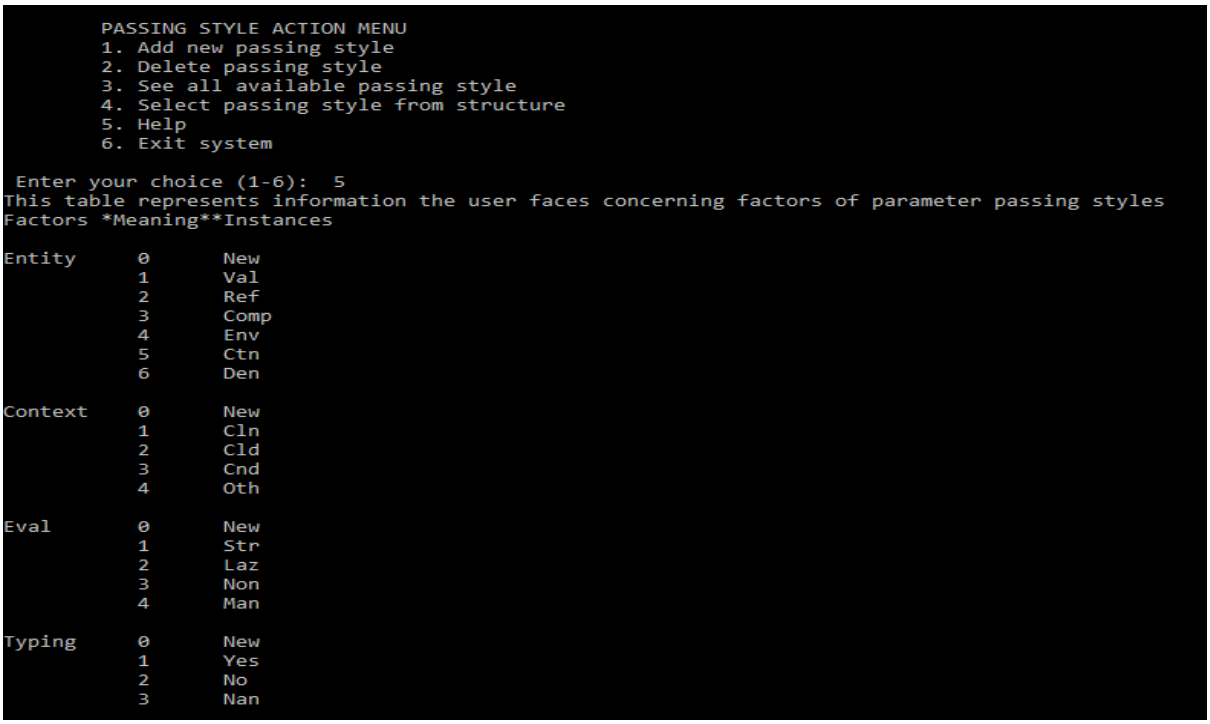

Fig. 4.3 Help menu

Figure 4.3 shows the help menu from which users can view possible instances for the various factors known to affect parameter passing.

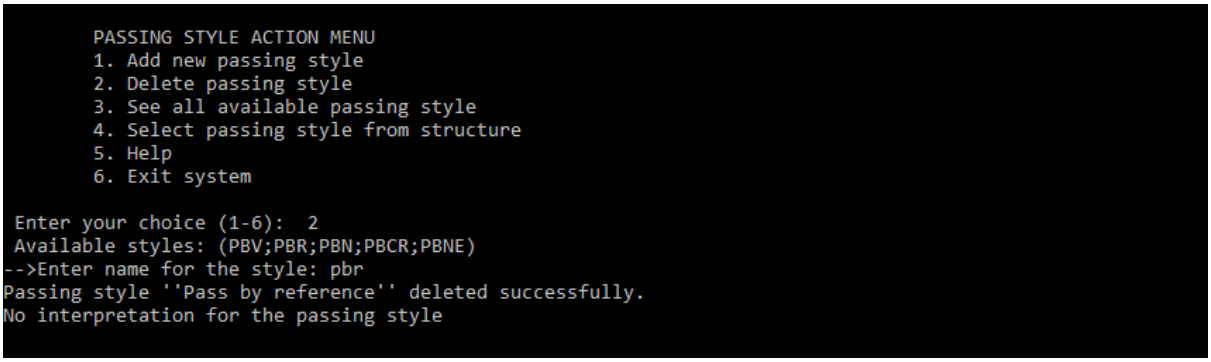

Fig.4.4 Deleting a passing style

Figure 4.4 demonstrates how a passing style is deleted (removed from the PSC). One selects a passing style he/she wants to delete by entering the passing style name. The name is then used to delete the passing style from the PSC. A success message is displayed upon a successful deletion. List of available passing styles one can select from is displayed before the passing style name is entered (like “pbr” for pass by reference) as shown in [Figure 4.4].

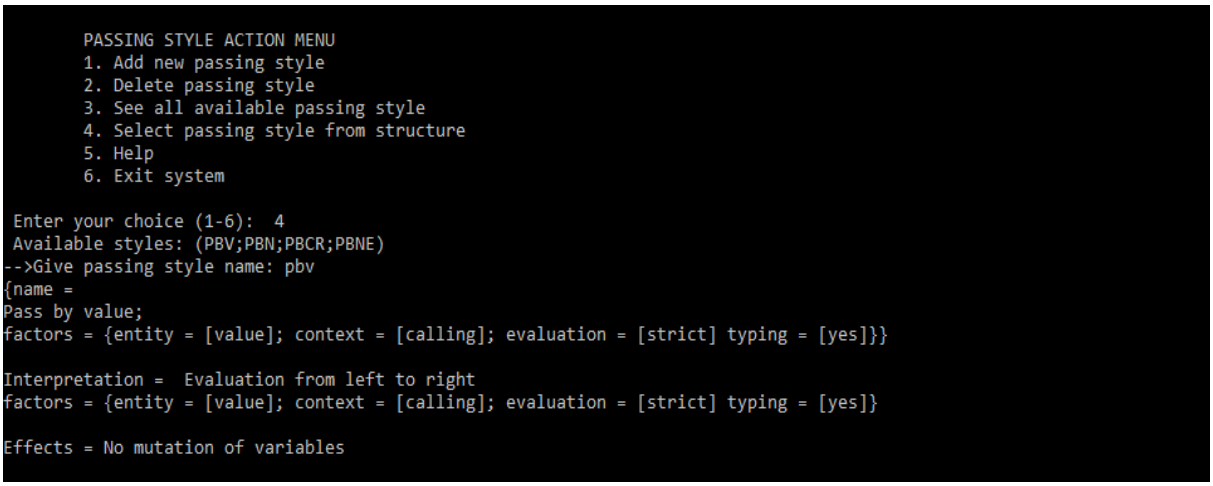

Fig. 4.5 Result after user selects passing style

Figure 4.5 shows the result obtained when a user selects a passing style from the passing style container. The passing style is selected by entering the passing style name. The name is used to display the properties of the style if the passing style exist in the PSC. The properties include: the passing style, factors that effects the style, interpretation over the style, and effects induced by values of factors of the style.

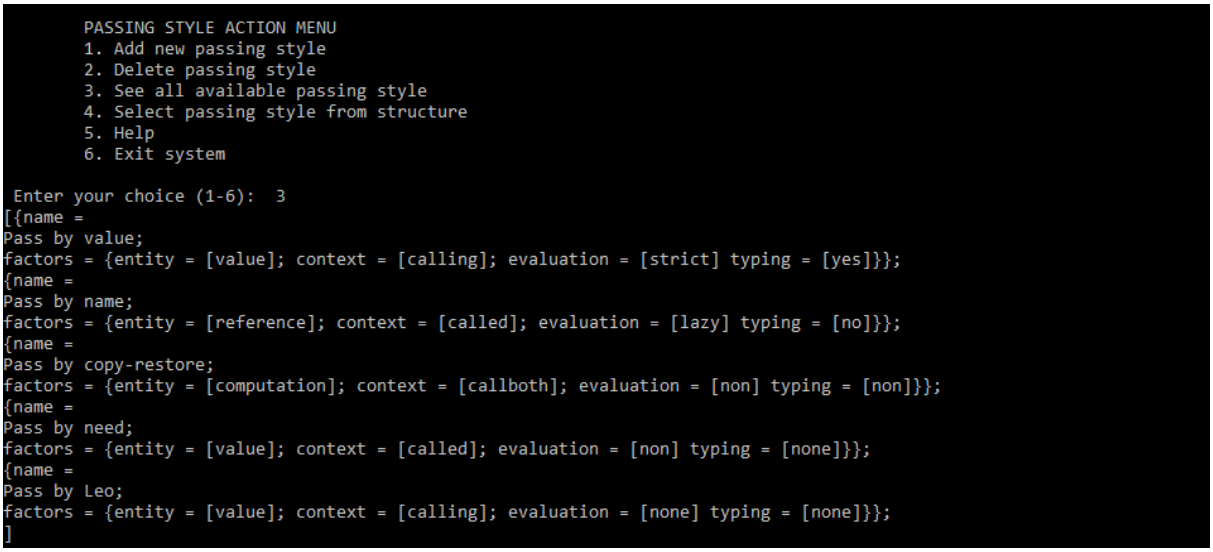

Fig. 4.7 All available styles

Figure 4.7 shows all available passing styles in the PSC after whatsoever operation that have been performed on the PSC.

All of the above Figures shows the results obtained when we ran the program. There are still some minor changes to be done to the code so that it runs with a better level of perfection. This project develops a PSC for parameter passing styles where operations to add, remove passing styles are defined on the PSC as shown in [Figure 4.1] to [Figure 4.5].

The developed PSC can be used to hold future parameter passing styles in a way that the parameter passing styles can easily be used and manipulated in the PSC. The developed PSC can also be integrated into programming languages. This integration can provide programming languages with the ability to implement the semantic of if not all but a greater amount of parameter passing styles.

The main challenges I faced was identifying suitable data structures to use for capturing the relationships amongst the main entities (or factors) involved in parameter passing. And also, implementing the flexibility of users to select known factor(s) of interest, and providing multiple instances for a selected factor.

As a scope of improvement on this project, the semantics (mathematical study of meaning) of a programming language can be used to implement the effects induced by a passing style including novel styles. This can be done by say generating code fragments that implements the effects induced by values of factors for the various passing styles rather than giving text description as done in this project.

# Chapter 5

## Conclusion

In this project, we developed a data structure for parameter passing that provides users with the tools to create, add, remove passing styles from the developed data structure, add an interpretation for a style, and to explore the various factors from which the behaviour of a style is defined. Through this work, we demonstrated how the passing style container is constructed from a collection of simple data structures, how it elements are organised and how they relate to one another. To ensure this structure is as flexible and efficient as possible, it effectively factors out entity, context, evaluation and typing which have been identified over the years as factors known to affect parameter passing styles. The analysis section identified permissible operations that can be carried out on the passing style container. The original objectives of this project as stated Project aims was to develop a data structure that holds various parameter passing styles, identify and group the various parameter passing styles. Also identify the relationships amongst parameter passing styles in terms of basic components. The original objective has been achieved.

# REFERENCES


[1] L. Moreau, «Introduction to Continuation,» *Introduction to Continuation,* pp. 10-51, 10 June 1994.

[2] C. Verela, «Typing, Parameter Passing and Lazy Evaluation,» *Typing, Parameter Passing and Lazy Evaluation,* pp. 11-25, 25 November 2014.

[3] J. ~~h~~Hickey, Introduction to Objective OCaml, California: Cambridge University Press, 2008.

[4] C. L. R. L. R. C. S. Thomas H. Corman, Introduction to Algorithms, Massachusetts: MIT Press, 2009.

# APPENDIX

1. **Algorithm to remove style from structure**

```
start
Read style_name   (*name of style to be deleted)
exit = false, temp = []   (*temporal empty list *)
REPEAT
Read head_of_list       (*first passing style in the structure*)
if style_name = head_of_list.name
 then
        list1 <--  remove_hd   hd  list   (*removing style if found*)
        list <-- Add list1 temp          (*merging list1 and temp*)
        exit = true                      (*condition to exit loop when style is found*)
 else
        temp <-- Add  hd  temp   (*insert passing styles that does not match to temp*)
        list <-- tail            (*tail list becomes new list for search*)
endif
UNTILL exit = true
Stop    (*exit operation*)
```

Fig.A: Algorithm to remove style

Figure A shows the algorithm for removing a passing style from the developed passing style structure. It uses the passing style name to find and delete the style from the passing style structure. It matches the name with names of passing styles in the structure. If name is found, the style is removed from the structure.

## 2. Implementation of the design algorithm

```
let rec usermind user_styles interpretation_list record effect_list =
  let () = display_menu ()
  in let () = print_string " Enter your choice (1-6): "
  in let opinion  = validate_input() in
  match (getOpinion opinion) with
 |"Add_style"-> let update = (insert_new_passing_style user_styles interpretation_list
effect_list    record)
 in let record = {update.record with ent = [5]; context = [5]; eval = [5]; typ = [5] }
                in  usermind update.styles update.interp record  update.effect
 |"Del_style"->let update = (delete_passing_style user_styles interpretation_list record
effect_list)
                     in   usermind  update.styles update.interp update.record update.effect
 |"Show_styles"->let update = (display_styles user_styles interpretation_list record effect_list)
                in  usermind  update.styles update.interp update.record update.effect
 |"Specific_style"->let update = (view_specific_passing_style user_styles interpretation_list
record effect_list)
               in  usermind  update.styles update.interp update.record update.effect
 |"Help"->let update = (user_facing_information user_styles interpretation_list record
effect_list) in
                usermind  update.styles update.interp update.record update.effect
  |_-> ()
```

Fig.B : Implementation of design algorithm

Figure B shows the implementation of the design algorithm in Figure 2.6. It provides users with a number of actions from which they can select from. Depending on the user's choice, a specific action is carried out except for the exit menu item which exits the system.

### 3. Function used to remove passing style(s)

```
let rec delete_passing_style styles interpretation_list record effect=
let () = print_string " Available styles: " in
let () = print_styles_name styles in    /*shows available styles in the list of styles*/
let () = print_string "\n-->Enter name for the style: " in  /*name of style to delete*/
let name = read_line() in
let rec removal styles interpretation_list=
match styles with
[] -> let () = print_string "Style don't exist." in []
|hd :: tl -> if String.lowercase_ascii (name) = String.lowercase_ascii (getName hd.name)
then let () = print_string "Passing style " in
let () = print_string """ in
let () = print_string (ful_name_style name)
in  let () = print_string """ in
remove_passing_style styles hd  /*removing passing style (hd) from the PSC (styles)*/
else
hd :: removal tl interpretation_list  /*recursive check if style not found in first check*/
in let styles = removal styles interpretation_list  /*Updating the passing style container*/
in let interp = delete_interpretation name interpretation_list /*delete style interpretation*/
in
{ styles = styles ; interp = interp; record = record ; effect = effect}
```

Fig.C : Implementation of deletion method

Figure C shows the implementation of the function used to remove a passing style from the passing style structure. The function [Fig.C ] gets the name of the style and delete the style from the passing style container if found. On deleting a passing style, the style interpretation and effects are also deleted.