

Generic Sorting in RESOLVE

Yu-Shan Sun

Dept. of Mathematics and Computer Science
Denison University
Granville OH, 43023, USA
Email: sun_s@denison.edu

Joan Krone

Dept. of Mathematics and Computer Science
Denison University
Granville OH, 43023, USA
Email: krone@denison.edu

Abstract—The RESOLVE vision of creating a verifying compiler[1] that allows software to be fully specified and verified, requires not only that the software be written with clean semantics, but that it contain mechanisms for reuse. To address the idea of reuse, here we emphasize generic sorting using the Prioritizer as the underlining concept. We address the challenge of providing proof of concept to show that our realizations are compatible not only with primitive data types, but also with complex data types whenever a definition of “Less Than or Equal To” is provided. This work illustrates RESOLVE’s generic goals[1,2].

I. INTRODUCTION

Because it is time consuming to fully specify software so that it can be mathematically verified, it is important that concepts and realizations can be reused in multiple settings. Current languages, such as Java and C++, provide limited generic capabilities, allowing a generic container data type to be parameterized by a limited kind of entry types and other primitives, such as integers.

However, there are no mechanisms in Java for the inclusion of mathematical definitions, such as a definition for “Less Than or Equal To” to accompany imported entry types. Moreover, C++ does not allow for multiple implementations of single header files. In any case, neither C++ nor Java support formal specifications or verification.

Our goal here is to put to the test the layered structure in RESOLVE that permits parameterization of concepts, realizations, and facilities by not only entry types, but mathematical definitions as needed, affirming the multiple realization and generic nature of RESOLVE.

Of course, we are limited by the current state of the compiler, and so may raise questions as to how the compiler can handle these generic requirements.

Sorting provides an ideal task for our goal, since the Prioritizer_Template, is parameterized not only by an entry type and a size, but by a mathematical definition as well. For each entry type, the specifier must include a definition for how the comparisons are to be made among entries.

II. SORTING REALIZATIONS

The key idea behind sorting is to organize items according to whatever definition is specified. When designing sorting algorithms for an Introductory Computer Science course, current languages force modification, rather than reuse, even for primitive types, to accommodate sorting of differ-

ent types. Although some components could be adapted to sort other data types, changing and adapting code may introduce errors. The goal of creating generic sorting algorithms based on a single concept requires not only that primitive data types can be sorted, but that user defined data types can be sorted as well. By using RESOLVE as the underlining language, we achieve abstraction and promote reuse by allowing the parameterization, not only of types, but definitions of sorting.

In the Prioritizer_Template[3], the Add_Entry procedure allows data to be entered, and the Change_Modes blocks addition of more data once the user finishes entering the data set. The Entry_Keeper is manipulated in different ways for sorting items. For Insertion Sort, the sorting can occur inside Add_Entry. When a new entry is added, it is inserted into its correct position. Both the Prioritizer Concept and the Insertion Sort Realization are shown in the appendix.

```

Concept Prioritizer_Template(type Entry; eval Max_Capacity: Integer;
                                def const (x: Entry) <= (y: Entry): B);
...
Family Entry_Keeper ⊆ Cart_Prod
    Entry_Count: Entry → ℕ;
    Accepting: B;
end;
    
```

Sorts such as Selection Sort and Quicksort use Add_Entry to insert the data in whatever order the user presents it. Sorting for these algorithms occurs more reasonably in Change_Modes once all the data has been added.

On the other hand, Heapsort will have to first construct the heap during Add_Entry. The Change_Mode just set the flag indicating the entry is complete. As the entries are removed, the algorithm must rebuild the heap in order to maintain its heap property. The current version of the RESOLVE compiler doesn’t yet handle mathematical definitions for the sorting algorithms. This prevents verification of these sorting realizations for now.

We point out that we must also include a replica operation as a parameter for our realizations, although such an inclusion is not peculiar only to sorting, but rather is fairly universal when working in RESOLVE.

We include code for Prioritizer_Template in Appendix A and Insertion Sort Realization in Appendix B. This and other implementations were done first for primitive entry

types to illustrate the multiple implementation capability in RESOLVE.

III. COMPLEX DATA TYPES

Next we turn to the challenge of making our sorting have not only the multiple implementation property, but the generic capability as well. We want to use a complex data type for which a special definition for comparing elements must accompany that type.

For example, if one wanted to sort student records, the definition of how to sort might be done on name or class rank or major, etc.

However, even sorting strings can be used for this challenge, since the simple “Less Than or Equal To” for integers is not sufficient and a definition for comparing strings must be made.

At the point of writing this paper, we are still working on using strings or some other complex data type as entry type. We need the compiler to recognize definitions as parameters in order to complete this work.

```
Family Student_Record  $\subseteq$  Cart_Prod
    StudentName: Str(Char);
    ClassYear: Str(N);
    Major: Str(Char);
    StudentID: Str(N);
end;
```

IV. CONCLUSION

Our handling of generic sorting based on the Prioritizer is currently a work in progress. There are still definitions and implementations to be written and verification to be completed. At this point, we see no reason why the generic capabilities of RESOLVE cannot be achieved once the compiler can recognize mathematical definitions as parameter and appropriate definitions has been completed for each component.

V. ACKNOWLEDGEMENT

I would like to thank Dr. Joan Krone for her contributions and suggestions in this project. A special thanks to Dr. Murali Sitaraman and Hampton Smith for providing updates of the compiler as well as clarifications in RESOLVE syntax and semantics. This research is made possible by the funding of Denison University Research Foundation.

VI. REFERENCES

1. J. Krone, W. F. Ogden, M. Sitaraman, and B. W. Weide, “Refocusing the Verifying Compiler Grand Challenge,” Technical Report RSRG-08-01, School of Computing, Clemson University, Clemson, SC 29634-0974, June 2008.
2. Krone, J., and Ogden, W.F. (2007). “Resolve 2007: Current State of Verification.” Proceedings of the RESOLVE 2007 Workshop. Clemson University, June 11-13, 2007.

3. Ogden, W.F., “The Proper Conceptualization of Data Structures,” Class Notes at O.S.U., 2002.

Appendix A: Prioritizer Template

(Source: <http://resolve.cs.clemson.edu/>)

```

Concept Prioritizer_Template(
    type Entry;
    evaluates Max_Capacity: Integer;

    definition LEQV(x,y: Entry): B
);
uses Std_Boolean_Fac, Std_Integer_Fac, Natural_Theory;
requires Max_Capacity > 0 and
    (for all x,y,z: Entry, if LEQV(x,y) and LEQV(y,z) then LEQV(x,z)) and
    (for all x,y: Entry, (LEQV(x,y) or LEQV(y,x)));
-----

Definition Total_Entry_Ct(P: Prioritizer): N =
    (Sum x: Entry, P.Entry_Count(x));

Type Family Prioritizer is modeled by Cart_Prod
    Entry_Count: Entry -> N;
    Accepting: B;
end;
exemplar P;
constraint Total_Entry_Ct(P) <= Max_Capacity;
initialization ensures P.Accepting = true and
    Total_Entry_Ct(P) = 0;
-----

Definition Is_Only_Difference(E: Entry, P1,P2: Prioritizer): B =
    P2.Entry_Count(E) = P1.Entry_Count(E) + 1 and
    (for all x: Entry,
        if x /= E then P2.Entry_Count(x) = P1.Entry_Count(x));

Definition Less(x,y: Entry): B = (LEQV(x,y) and not LEQV(y,x));

Operation Add_Entry(alternates E: Entry; updates P: Prioritizer);
    requires Total_Entry_Ct(P) < Max_Capacity and P.Accepting = true;
    ensures Is_Only_Difference(#E, #P, P) and P.Accepting = true;

Operation Change_Mode(updates P: Prioritizer);
    ensures P.Accepting = not #P.Accepting and
        P.Entry_Count = #P.Entry_Count;

Operation Remove_A_Smallest_Entry(replaces S: Entry;
    updates P: Prioritizer);
    requires P.Accepting = false and Total_Entry_Ct(P) > 0;
    ensures P.Accepting = false and Is_Only_Difference(S, P, #P) and
        (for all E: Entry, if Less(E, S) then #P.Entry_Count(E) = 0);

Operation Remove_Any_Entry(replaces E: Entry; updates P: Prioritizer);
    requires Total_Entry_Ct(P) > 0;
    ensures Is_Only_Difference(E, P, #P) and P.Accepting = #P.Accepting;

Operation Is_Accepting(restores P: Prioritizer): Boolean;
    ensures Is_Accepting = (P.Accepting);

Operation Total_Entry_Count(restores P: Prioritizer): Integer;
    ensures Total_Entry_Count = Total_Entry_Ct(P);

Operation Rem_Capacity(restores P: Prioritizer): Integer;
    ensures Rem_Capacity = (Max_Capacity - Total_Entry_Ct(P));

Operation Clear(clears P: Prioritizer);

end Prioritizer_Template;

```

Appendix B: Insertion Sort Ordering

(Source: <http://resolve.cs.clemson.edu/>)

```

Realization Insertion_Ordering_Realiz (
  operation Are_Ordered(restores x,y: Entry): Boolean;
    ensures Are_Ordered = (LEQV(x,y));
  ) for Prioritizer_Template;
uses Set_Theory;

Definition Array_In_Order(P: Prioritizer): B =
  For all i,j: Z,
    if 1 <= i <= j <= P.Index_of_Smallest
      then LEQV(P.Entry_Seq(j), P.Entry_Seq(i));

Definition Array_Entry_Ct(P: Prioritizer, x: Entry): N =
  |{i: Z where 0 < i <= P.Index_of_Smallest, P.Entry_Seq(i) = x}|;

Type Prioritizer = Record
  Entry_Seq: Array 1..Max_Capacity of Entry;
  Index_of_Smallest: Integer;
  Accepting_Flag: Boolean;
end;

correspondence Conc.P.Accepting = P.Accepting_Flag and
  (for all x: Entry, Conc.P.Entry_Count(x) = Array_Entry_Ct(P, x));

initialization
  P.Accepting_Flag := True();
  P.Index_of_Smallest := 0;
end;

Procedure Add_Entry(alters E: Entry; updates P: Prioritizer);
  Var Next_Index: Integer;
  Var Next_Entry: Entry;
  Var B: Boolean;
  P.Index_of_Smallest := P.Index_of_Smallest + 1;
  Next_Index := P.Index_of_Smallest;
  B := True();
  While B = True()
    changing Next_Index, Next_Entry, P.Entry_Seq;
    maintaining 1 <= Next_Index <= P.Index_of_Smallest and
      (for all i: Z,
        if 1 <= i and i < Next_Index
          then P.Entry_Seq(i) = #P.Entry_Seq(i) and
            (if Next_Index <= i < P.Index_of_Smallest
              then (P.Entry_Seq(i+1) = #P.Entry_Seq(i) and
                Less(#P.Entry_Seq(i), E)));
        decreasing |Next_Index|;
      do
        Next_Index := Next_Index - 1;

        if Next_Index = 0 then
          B := False();
        else
          Next_Entry := P.Entry_Seq(Next_Index);
          if Are_Ordered(E, Next_Entry) = True() then
            P.Entry_Seq(Next_Index) := Next_Entry;
            B := False();
          else
            P.Entry_Seq(Next_Index + 1) := Next_Entry;
          end;
        end;
      end;
    end;
  P.Entry_Seq(Next_Index + 1) := E;
end Add_Entry;

Procedure Change_Mode(updates P: Prioritizer);
  P.Accepting_Flag := Not(P.Accepting_Flag);
end Change_Mode;

```

```
Procedure Remove_A_Smallest_Entry(replaces S: Entry;
                                  updates P: Prioritizer);
    S := P.Entry_Seq(P.Index_of_Smallest);
    P.Index_of_Smallest := P.Index_of_Smallest - 1;
end Remove_A_Smallest_Entry;

Procedure Remove_Any_Entry(replaces E: Entry; updates P: Prioritizer
    E := P.Entry_Seq(P.Index_of_Smallest);
    P.Index_of_Smallest := P.Index_of_Smallest - 1;
end Remove_Any_Entry;

Procedure Is_Accepting(preserves P: Prioritizer): Boolean;
    Is_Accepting := (P.Accepting_Flag);
end Is_Accepting;

Procedure Total_Entry_Count(preserves P: Prioritizer): Integer;
    Total_Entry_Count := P.Index_of_Smallest;
end Total_Entry_Count;

Procedure Rem_Capacity(preserves P: Prioritizer): Integer;
    Rem_Capacity := (Max_Capacity - P.Index_of_Smallest);
end Rem_Capacity;

Procedure Clear(clears P: Prioritizer);
    P.Accepting_Flag := True();
    P.Index_of_Smallest := 0;
end Clear;

end Insertion_Ordering_Realiz;
```