Specification and Verification of C3 linearization algorithm

Miguel Flor[1], António Ravara[2], and Mário Pereira[2]

NOVA LINCS, Nova School of Science and Technology
m.flor@campus.fct.unl.pt, {aravara, mjp.pereira}@fct.unl.pt

Abstract. Multiple inheritance is common in modern programming languages, and the C3 linearization algorithm is widely adopted to ensure a consistent and intuitive method resolution order (MRO). Despite its common use, particularly in Python, the algorithm is non-trivial, and rigorous formal specification and verification remain necessary. In this paper, we present a mathematical formalization of the C3 algorithm and initiate its verification using OCaml, Why3, and Cameleer. Our work proves the termination and partial verification of the core components of the algorithm. While complete verification of the linearization process is ongoing, our contributions represent an essential step toward a complete proof of C3's verification.

1 Introduction

The development of large software systems often requires object-oriented programming languages that support multiple inheritance and provide a predictable method resolution order (MRO). That's why the C3 linearization algorithm is used in many programming languages Python 2.3[1], Perl[2], Solidity[3], and many others. And exactly because of its widespread use, there is a need for an implementation that is verified, in this paper the verification is done in OCaml using Why3 and Cameleer.

1.1 Why C3?

C3 algorithm is widely used for producing a consistent and predictable MRO when conflicts arise in multiple inheritance scenarios. The algorithm uses three core properties to ensure that the MRO is consistent (these properties are explained in section 2.2):

- Extended Precedence Graph (EPG)
- Local Precedence Order (LPO)
- Monotonicity

To understand the algorithm consider the class hierarchy in Fig. 2, where class A inherits from both B and C. The C3 algorithm resolves the conflict ensuring that the MRO respects the order of inheritance (reading from left to right), resulting in an MRO of (A, B, C, D).

1.2 Contributions

To reach the goal of a verified implementation of the C3 linearization algorithm, we present the following contributions:

- A mathematical formal specification of the C3 linearization algorithm, which includes the properties that the algorithm must satisfy.
- An Ocaml implementation of the C3 linearization algorithm.
- A partial verification of the implementation using Why3 and Cameleer, demonstrating essential correctness properties and termination.

2 Background

We will first examine how multiple inheritance affects MRO algorithms, then examine the principles of the C3 linearization algorithm, followed by a brief overview of cameleer.

2.1 Influence of Multiple Inheritance on MRO Algorithms

MRO is a linear extension of some class hierarchy that induces a total order on the classes in the hierarchy[4].

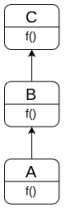

Fig. 1. Simple graph

For the example above (Fig. 1) a logical MRO could be (A, B, C), this means that when the function f is executed from the class A the implementation of A will be prioritized over B, and the implementation of B will be prioritized over C.
However, the MRO of a class hierarchy is not always that straightforward, and conflicts can start to arise when we introduce multiple inheritance. To demonstrate this, consider the following example, that represents a known problem called the diamond problem[5].

In this example (Fig. 2), a conflict emerges when determining the MRO of class A, since A inherits from both B and C. This raises the question: Which implementation, B's or C's, does A inherit from? This is where the C3 linearization algorithm comes into play, as it provides a consistent and predictable way to resolve this conflict, the algorithm says that we need to prioritize B over C, because B comes first in the class hierarchy, and therefore the MRO of A will be (A, B, C, D).

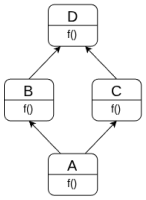

Fig. 2. Diamond problem graph

2.2 C3 Linearization Algorithm

The C3 algorithm produces an MRO that is consistent with EPG, LPO, and monotonicity[6] (all of these properties are formally defined in section3.2).

EPG To understand the EPG, consider the following graph (Fig. 3). For this graph instead of ordering the classes from left to right, they are arranged by the weights of the directed edges, from smallest to largest.

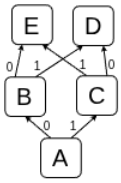

Fig. 3. Inconsistent EPG

This graph is inconsistent with the EPG, as the parent classes of B and C are ordered differently. A valid class hierarchy requires that for each class, the parent classes are ordered in the same way.

LPO Consider the graph in Fig. 2. An MRO that is consistent needs to respect the priority of the classes locally at each inheritance point; therefore, for this case, it cannot be (A, C, B, D).

Monotonicity If an MRO algorithm is consistency with monotonicity, it means that if a new class is added to the hierarchy the order of the previous MRO is preserved.

2.3 Cameleer and Why3

Cameleer[7] is an automated tool for deductive verification in OCaml. It uses comments written in GOSPEL[8] (Generic OCaml Specification Language) to

specify the properties of functions, and then leverages Why3[9] to verify these properties.
With GOSPEL, we can write pre-conditions, post-conditions, invariants, and variants using comments that start with a @ and are attached to a specific OCaml function. Take the OCaml function below as an example:

```
1  let rec filter_heads (l: 'a list list) =
2      match l with
3      | [] -> []
4      | (h :: _) :: t -> h :: filter_heads t
5      | [] :: t -> filter_heads t
6    (*@ r = filter_heads l
7      ensures forall y. List.mem y r ->
8        (exists x h t. List.mem x l /\ h::t = x /\ h = y)
9      ensures (forall x h t. List.mem x l /\ x = h::t -> List.mem h r)
10     variant l
11   *)
```

In the function `filter_heads` takes a list of lists as argument and returns a list of the heads of each sublist, the GOSPEL comments two post-conditions and a variant. The first post-condition (lines 7 and 8) states that for every element y in the result list r, exists a sublist x in the input list l such that the head of x is equal to y. The second post-condition (lines 9) states that for every sublist x in the input list l, the head of x is in the result list r. And the variant (line 10) ensures the termination of the function by stating that the input list l is decreasing in size with each recursive call.
To run the verification, we can use the command `cameleer <file name>.ml`, which will open the Why3 IDE, like the one in Fig.4.

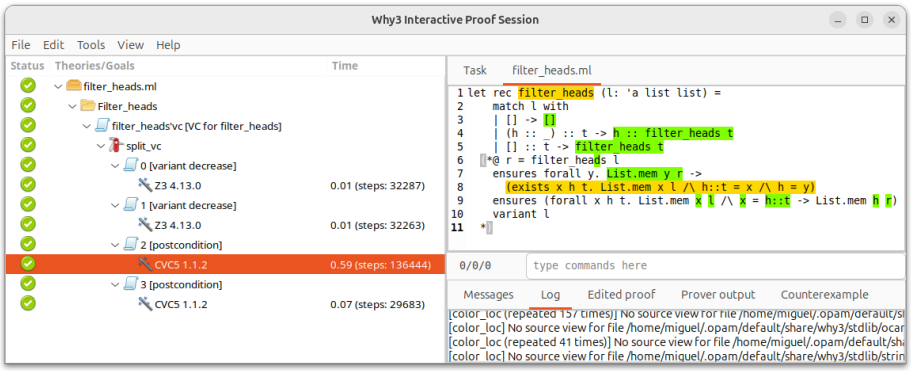

Fig. 4. `filter_heads.ml` in Why3 IDE

As demonstrated by the green checkmark on the left side of the screen, the proof has been successfully verified. In the IDE, we proved these goals us-

ing provers, for our work we used three main provers, Alt-Ergo 2.6.0[10], Z3 4.13.0[11], and CVC5 1.1.2[12] and for each goal, we can select which prover to use. To automate the proof Why3 provides a list of proof strategies including `Split_vc`, `Auto_level_0`, `Auto_level_1`, `Auto_level_2`, and `Auto_level_3`. We also have commands to assist in the proof process, these commands are used to manipulate terms and hypotheses. All the commands and strategies are documented in Why3 documentation [13]. Below is a list of commands that we utilized in our work:

- `instantiate`
- `induction_arg_ty_lex`
- `intros`
- `destruct`
- `destruct_term`
- `unfold`
- `split_vc`

3 Formal Specification of C3 Linearization Algorithm

3.1 Ingredients

Let $\mathcal{C}$ be a finite set of symbols, ranged over by $C_0, C_1, \ldots, C_n$, possibly primed, which we refer to as *classes*, and let $(\mathcal{C}, <_c)$ be an ordered set of classes.

Consider $\mathcal{P} \subseteq (\mathcal{C}, <_c)$ and $\mathcal{D} \subseteq \mathcal{C} \times \wp(\mathcal{P})$.

The purpose of this document is to present the MRO algorithm, which maps each $\mathcal{C}$ with an ordered set of classes, $\mathcal{P}$. We show first the envisaged properties, then define it, and finally prove the definition ensures those properties.

3.2 Properties

Let $D = \langle C, \{P_1, \ldots, P_n\}\rangle \in \mathcal{D}$, with $n \in \mathbb{N}_0$.
Let $\texttt{MRO}(C) = \{C, C_1 \ldots, C_m\}$, with $m \in \mathbb{N}_0$.

Consistency with the EPG This property requires that $\{P_1, \ldots, P_n\} \subseteq \{C_1, \ldots, C_m\}$.

Consistency with the LPO For $m, n \geq 2$.

$$\forall\, i, j \ \big(0 \leq i < j \leq n \implies \exists\, p, q \ \big(0 \leq p < q \leq m \wedge C_p = P_i \wedge C_q = P_j\big)\big)$$

Consistency with monotonicity Let $D', D'' \in \mathcal{D}$
If $C \in \pi_2(D') \setminus \pi_2(D'')$, then
$\exists p, q.\ 0 < p < q \leq m \ \wedge\ C_p = C \ \wedge\ C_q = \pi_1(D'')$.

3.3 Functions

Let $(\mathcal{C}, <_c)^*$ be a sequence of ordered sets of classes.
Let $L = (L_1, \ldots, L_n)$, $L \in (\mathcal{C}, <_c)^*$
Let $C \in \mathcal{C}$

Remove $\texttt{remove} : (\mathcal{C}, <_c)^* \times \mathcal{C} \Rightarrow (\mathcal{C}, <_c)^*$

$$\texttt{remove}((\ ), C) = (\ )$$
$$\texttt{remove}(l :: L, C) = l \setminus \{C\} :: \texttt{remove}(L, C)$$

Merge $\texttt{merge} : (\mathcal{C}, <_c)^* \Rightarrow (\mathcal{C}, <_c)$

$$\texttt{merge}(L) = \begin{cases} \{C\} \cup \texttt{merge}(\texttt{remove}(L, C)), & \text{if (1), (2), (3)} \\ \text{fail}, & \text{otherwise} \end{cases}$$

where:

$$\begin{aligned} &(1) \quad \exists k \in [\![1, n]\!], L_k \neq \emptyset \wedge C = \texttt{head}(L_k) \\ &(2) \quad \forall j < k, C \neq \texttt{head}(L_j) \\ &(3) \quad \forall i \in [\![1, n]\!], C \notin \texttt{tail}(L_i) \end{aligned}$$

C3 Linearization $\texttt{c3linearization} : \mathcal{D} \Rightarrow (\mathcal{C}, <_c)$
Let $D = \langle C, P \rangle$ where $D \in \mathcal{D}$
Let $D' = (D_1, D_2, \ldots, D_{|P|})$, such that
$\forall P_i \in P,\ \exists D_i \in \mathcal{D}$ where $D_i = \langle P_i, P' \rangle$ where
$i \in [\![1, |P|]\!]$.
Let `C3linearization` be denoted as `cl` for brevity.

$$\texttt{cl}(D) = \begin{cases} \{C\} & \text{if } P = \emptyset \\ \{C\} \cup \texttt{merge}\left((\texttt{cl}(D_i))_{D_i \in D'}\ ,\ P\right) & \text{otherwise} \end{cases}$$

4 Verification of C3 Linearization Algorithm

In this project, we have two versions of the code, an original one and a version with some modifications to help the proof, and the code is divided into three parts: the `remove` where the remove of the head occurs, the `merge` where a new list created follows the same order has the order lists and finally the `c3_inearization` that will receive a class as argument and, in the version for the proof, a universe, a variable that contains all the class above in the hierarchy of that class.

4.1 Module CLASS

The code provided below represents a module that defines a class within the hierarchy:

```
4  module type CLASS = sig
5
6    type t
7
8    val eq : t -> t -> bool
9    (*@ b = eq x y
10         ensures b <-> x = y *)
11
12   val get_parents : t -> t list -> t list
13   (*@ l = get_parents c u
14     requires distinct u
15     requires Sequence.mem c u
16     ensures not (List.mem c l)
17     ensures distinct l
18     ensures forall i. Sequence.mem i l -> Sequence.mem i u
19   *)
20   val to_string : t -> string
21
22 end
```

The module was written to be as flexible as possible, so it could be easily implemented. Later, it proved difficult to prove (as explained in Section **??**).

The function `get_parents` takes a class and a list of lists as arguments and returns a list of parents of the class. The only change from the original implementation is the list of lists as an argument in the function `get_parents`, which was needed to ensure the hierarchy remained acyclic, passing as universe only the classes above in the hierarchy of the class.

4.2 remove function

Every time the function merge finds a head that has the three conditions (section 3.3) satisfied, the remove function is called. This function removes a given head from all the list of lists, and it is defined as follows:

```
286    let rec remove_aux (l: C.t list list) (e: C.t) : C.t list list =
287      match l with
288      | [] -> assert false
289      | [a] -> [remove_head e a]
290      | a :: r -> remove_head e a :: remove_aux r e
291    (*@ r = remove l e
292        requires l <> []
293        requires forall e. Sequence.mem e l -> distinct e
294
295        ensures r <> []
296        ensures forall i. 0 <= i < List.length r ->
```

```
297           not (has_head (Sequence.get r i) e)
298
299         ensures List.length l = List.length r
300         ensures forall i : int. 0 <= i < List.length l ->
301           is_removed (Sequence.get l i) (Sequence.get r i) e
302
303         variant  l *)
304     let remove (l: C.t list list) (e: C.t) : C.t list list =
305       remove_aux l e
306     (*@ r = remove l e
307       requires l <> []
308       requires  forall i. 0 <= i < List.length l ->
309          distinct (Sequence.get l i)
310       requires exists i. 0 <= i < List.length l /\
311        has_head (Sequence.get l i) e
312       requires forall i. 0 <= i < List.length l ->
313          not (List.mem e (tail (Sequence.get l i)))
314
315       ensures forall i. 0 <= i < List.length r ->
316          not (List.mem e (Sequence.get r i))
317       ensures forall i. 0 <= i < List.length r ->
318          is_removed (Sequence.get l i) (Sequence.get r i) e
319       ensures forall i. Sequence.mem i r -> distinct i
320       ensures sum_lengths r < sum_lengths l
321       ensures List.length r = List.length l
322       ensures r <> []
323     *)
```

The GOSPEL specification for `remove` includes preconditions and postconditions that aren't strictly needed to prove what `remove` itself does. But since `remove` is called inside `merge`, and `merge` relies on stronger guarantees, the extra conditions make it easier to prove `merge` correct. So, even if removal could be verified with fewer assumptions, we added specific pre-conditions to simplify the verification of the functions that depend on it.
This function defines two major postconditions:

- The argument `e` is not present in the any of the lists of the argument `l` after the removal.
- The sum of the lengths of the lists of `l` are bigger than the sum of the lengths of the lists of the result.

The last postcondition is pivotal to prove the termination of the function `merge`. To help the proof four lemmas were added to the code:

- `list_seq_mem`: This lemma state that using both modules List or Sequence should produce the same result.
- `is_removed_not_mem`: If an element is removed from the head (if the head is equal to the element) of a list and the tail does not contain that element then the element is not in the list.

- `is_removed_length_for_lists`: For all two lists of lists l and r, and for each index i, the index 0 is removed from $l[i]$ if $l[i][0] = e$ to produce $r[i]$, then the length of each $r[i]$ is less than or equal to the length of the corresponding $l[i]$.
- `is_removed_length`:If we remove the head of a list l if the head is equal to e and the head in l is equal to e then the length of l decreases.
- `sum_lengths_of_lists_l_e`: For all two lists of lists l and r of the same length, for each index i, the length of $l[i]$ is greater than or equal to the length of $r[i]$, then the sum of the lengths of l is greater than or equal to the sum of the lengths of r.
- `sum_lengths_of_lists_l`: For all two lists of lists l and r of the same length, for each index i, the length of $l[i]$ is greater than or equal to the length of $r[i]$ and for one j, length of $l[j]$ is greater than $r[j]$, then the sum of the lengths of l is greater than the sum of the lengths of r.

In the image (Fig. 5), we can see that the function was validated in the Why3 IDE.

Fig. 5. `remove` proof in Why3 IDE

4.3 Detect order inconsistencies in a list of lists

The function `merge` is the most complex function in the C3 linearization algorithm; this function receives a list of lists and returns a list where the order that each class has with each other respects in every list of the input.
To detect this, a precedence graph builds from the relations between the classes in the lists. For example, if we have the list of lists $((A, B, C), (C, D), (D, A))$,

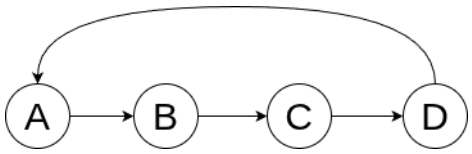

Fig. 6. Cyclic precedence graph

the graph in Fig. 6 is built. We can see that the graph is cyclic, and therefore, the order of the classes is inconsistent, and a linearization cannot be produced. And it must exist an element that for every list of a list, this element is not in the tail of the list and it exists a head from a list that is equal to the element. To detect the acyclicity of a graph, the following predicate was written:

```
(*@ predicate acyclic_precedence_graph (lins: C.t list list) =
  forall c.
    not (
      exists path: C.t list.
        List.length path > 1 /\
        Sequence.get path 0 = c /\
        Sequence.get path (List.length path - 1) = c /\
        (forall i.
          0 <= i /\ i < (List.length path) - 1 ->
            exists lin: C.t list, j.
              List.mem lin lins /\
              List.length lin > 1 /\
              0 <= j /\ j < (List.length lin) - 1 /\
              Sequence.get lin j = Sequence.get path i /\
              Sequence.get lin (j+1) = Sequence.get path (i+1)
        )
    ) *)
```

This predicate ensures that for the graph to be acyclic, no path exists in the graph where the first class is the same as the last class. The lemmas created from this predicate are not proven in this project and assumed by postconditions and preconditions of the `merge` and `c3_linearization` functions.

References

[1] *The Python 2.3 Method Resolution Order*. https://docs.python.org/3/howto/mro.html. (Visited on 06/25/2025).
[2] *Mro - Method Resolution Order - Perldoc Browser*. https://perldoc.perl.org/mro. (Visited on 06/25/2025).
[3] *Language Influences — Solidity 0.8.31 Documentation*. https://docs.soliditylang.org/en/latest/language-influences.html. (Visited on 06/25/2025).

[4] Florent Hivert and Nicolas M. Thiéry. "Controlling the C3 Super Class Linearization Algorithm for Large Hierarchies of Classes". In: *Order* 41.1 (Apr. 2024), pp. 83–98. ISSN: 1572-9273. DOI: 10.1007/s11083-022-09607-5. (Visited on 06/25/2025).

[5] Monday Eze, Charles Okunbor, and Umoke Chukwudum. "Studies in Object-Oriented Programming Backbone Implementations". In: *Global Journal of Engineering and Technology Advances* 8.3 (Sept. 2021), pp. 020–031. ISSN: 25825003. DOI: 10.30574/gjeta.2021.8.3.0119. (Visited on 06/25/2025).

[6] Kim Barrett et al. "A Monotonic Superclass Linearization for Dylan". In: *Proceedings of the 11th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. OOPSLA '96. New York, NY, USA: Association for Computing Machinery, Oct. 1996, pp. 69–82. ISBN: 978-0-89791-788-9. DOI: 10.1145/236337.236343. (Visited on 06/25/2025).

[7] Mário Pereira and António Ravara. "Cameleer: A Deductive Verification Tool for OCaml". In: *Computer Aided Verification*. Ed. by Alexandra Silva and K. Rustan M. Leino. Cham: Springer International Publishing, 2021, pp. 677–689. ISBN: 978-3-030-81688-9. DOI: 10.1007/978-3-030-81688-9_31.

[8] Arthur Charguéraud et al. "GOSPEL—Providing OCaml with a Formal Specification Language". In: *Formal Methods – The Next 30 Years*. Ed. by Maurice H. ter Beek, Annabelle McIver, and José N. Oliveira. Cham: Springer International Publishing, 2019, pp. 484–501. ISBN: 978-3-030-30942-8. DOI: 10.1007/978-3-030-30942-8_29.

[9] Jean-Christophe Filliâtre and Andrei Paskevich. "Why3 — Where Programs Meet Provers". In: *Programming Languages and Systems*. Ed. by Matthias Felleisen and Philippa Gardner. Berlin, Heidelberg: Springer, 2013, pp. 125–128. ISBN: 978-3-642-37036-6. DOI: 10.1007/978-3-642-37036-6_8.

[10] OcamlPro SAS. *The Alt-Ergo SMT Solver by OCamlPro*. https://alt-ergo.ocamlpro.com. (Visited on 07/03/2025).

[11] *Z3Prover/Z3*. Z3 Theorem Prover. July 2025. (Visited on 07/03/2025).

[12] *About Cvc5*. https://cvc5.github.io/. (Visited on 07/03/2025).

[13] *12. Technical Informations — Why3 1.8.1 Documentation*. https://www.why3.org/doc/technical.html. (Visited on 07/03/2025).