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 hierarchical inheritance is common among many programming languages, and to produce a consistent and intuitive method resolution order (MRO) of the class hierarchy, the C3 linearization algorithm is one of the most widely used. However, the algorithm is not trivial, and because its widespread use, there is a need for a formal specification and verification of this algorithm. In this paper, we present a mathematical formal specification of the C3 algorithm, and its verification in OCaml code using Why3 and Cameleer.

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, which is verified using Why3 and Cameleer.

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].

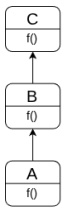

Fig. 1. Simple graph

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).

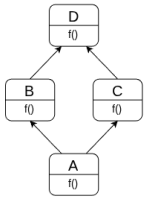

Fig. 2. Diamond problem graph

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.

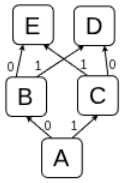

Fig. 3. Inconsistent EPG

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.

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 $m \in \mathbb{N}_0$.

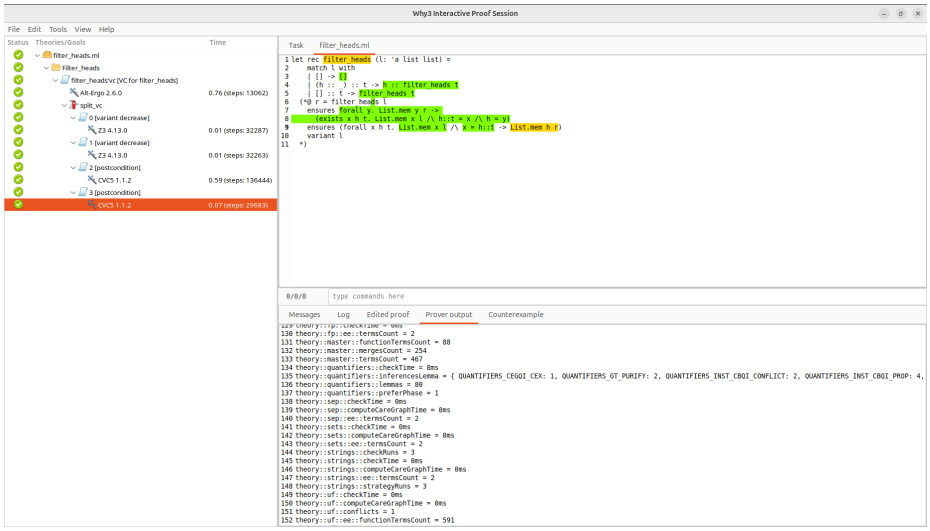

Fig. 4. `filter_heads.ml` in Why3 IDE

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.

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

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.