Specification and Verification of the C3 linearization algorithm

Miguel Flor, Mário Pereira, and António Ravara

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

Abstract. Multiple inheritance is common in modern programming languages, and the C3 linearization algorithm [1] is widely adopted to ensure a consistent and intuitive method resolution order (MRO) [2]. Despite its common use, particularly in Python, the algorithm is non-trivial, and to gain confidence of its correctness, 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.

Keywords: C3 linearization · multiple inheritance · method resolution order · formal verification · OCaml · Why3 · 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 such as Python 2.3 [3], Perl [2], Solidity [4], and many others. 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

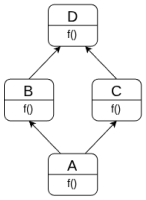

Fig. 1. Diamond problem graph

To understand the algorithm consider the class hierarchy in Fig. 1, 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 (section 3).
- An Ocaml implementation of the C3 linearization algorithm (section 4).
- A partial verification of the implementation using Why3 and Cameleer, demonstrating essential correctness properties and termination(section 4).

Both the formal specification and the implementation are available in the repository [5].

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 [6].

For the example above (Fig. 2) 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 example in Fig. 1, that represents a known problem

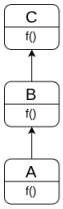

Fig. 2. Simple graph

called the diamond problem [7]. In this example, 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 states 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).

2.2 C3 Linearization Algorithm

The C3 algorithm produces an MRO that is consistent with EPG, LPO, and monotonicity [1] (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.

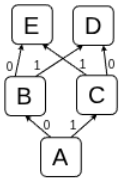

Fig. 3. Inconsistent EPG

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. 1. 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 consistent with monotonicity, it means that if a new class is added to the hierarchy the order of the previous MRO is preserved. For example, in Fig. 4, if the MRO of the first hierarchy is (A, B, C, D),

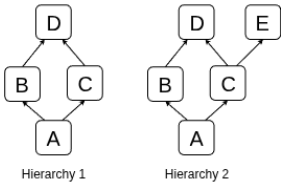

Fig. 4. Two class hierarchies, on of them has one more class

then the MRO of the second hierarchy can't be (A, C, B, D, E), because the order of B and C is not preserved.

2.3 Cameleer and Why3

Cameleer [8] is an automated tool for deductive verification in OCaml. It uses comments written in GOSPEL [9] (Generic OCaml Specification Language) to specify the properties of functions and then leverages Why3 [10] 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 that we developed as part of the C3 algorithm 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   *)
```

The function `filter_heads` takes a list of lists as argument and returns a list of the heads of each sublist, the GOSPEL comments specify two postconditions and a variant. The first postcondition (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 postcondition (lines 9) states that for every sublist x in the input list l, the head of x is in the result list r. 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.5.

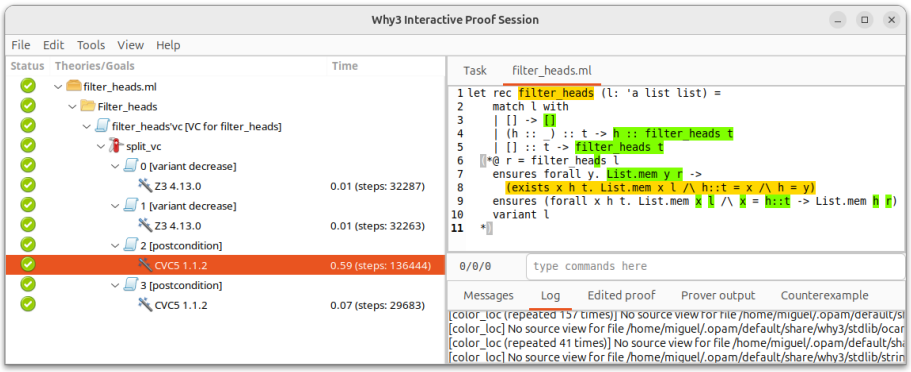

Fig. 5. `filter_heads.ml` in Why3 IDE

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 using provers. For our work we used three main provers, Alt-Ergo 2.6.0 [11], Z3 4.13.0 [12], and CVC5 1.1.2 [13] 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 [14]. Below is a list of commands that we utilized in our work:

- `instantiate`: Instantiates a quantified hypothesis with specific values.
- `induction_arg_ty_lex`: Performs induction based on the type and a lexicographic ordering of arguments.
- `intros`: Moves universally quantified variables and assumptions into the context.
- `destruct`: Breaks a compound hypothesis (e.g., conjunction, disjunction) into its components.
- `destruct_term`: Performs case analysis on algebraic data types, creating one subgoal per constructor.
- `unfold`: Expands a function or definition inline for easier reasoning.
- `split_vc`: Splits verification conditions into independent subgoals.
- `inline_goal`: Replaces a function application with its definition inside the goal.

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 part 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 $\{P_1, \ldots, P_n\} \subseteq \{C_1, \ldots, C_m\}$.

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

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

$$\texttt{remove}((\ ), C) = (\ )$$

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

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

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

where:

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

$\texttt{c3linearization} : \mathcal{D} \to (\mathcal{C}, <_c)$
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. The code is divided into three parts: 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 revealed difficult to prove (as explained in Section 6).

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, taking 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:

```
290   (* Removes from a list of lists of classes (l) an element (e) *)
291   let rec remove_aux (l: C.t list list) (e: C.t) : C.t list list =
292     match l with
293     | [] -> assert false
294     | [a] -> [remove_head e a]
295     | a :: r -> remove_head e a :: remove_aux r e
296   (*@ r = remove l e
297       requires l <> []
298       requires forall e. Sequence.mem e l -> distinct e
299
300       ensures r <> []
301       ensures forall i. 0 <= i < List.length r ->
302          not (has_head (Sequence.get r i) e)
303
304       ensures List.length l = List.length r
305       ensures forall i : int. 0 <= i < List.length l ->
306          is_removed (Sequence.get l i) (Sequence.get r i) e
307
308       variant  l *)
309   let remove (l: C.t list list) (e: C.t) : C.t list list =
310     remove_aux l e
311   (*@ r = remove l e
312     requires l <> []
313     requires  forall i. 0 <= i < List.length l ->
314        distinct (Sequence.get l i)
315     requires exists i. 0 <= i < List.length l /\
```

```
316         has_head (Sequence.get l i) e
317       requires forall i. 0 <= i < List.length l ->
318          not (List.mem e (tail (Sequence.get l i)))
319
320       ensures forall i. 0 <= i < List.length r ->
321          not (List.mem e (Sequence.get r i))
322       ensures forall i. 0 <= i < List.length r ->
323          is_removed (Sequence.get l i) (Sequence.get r i) e
324       ensures forall i. Sequence.mem i r -> distinct i
325       ensures sum_lengths r < sum_lengths l
326       ensures List.length r = List.length l
327       ensures r <> []
328     *)
```

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 Fig. 6, we can observe that the function was validated in the Why3 IDE.

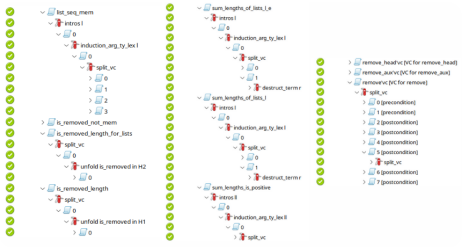

Fig. 6. `remove` proof in Why3 IDE

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 is built from the relations between the classes in the lists. For example, if we have the list of lists $((A, B, C), (C, D), (D, A))$, the graph in Fig. 7 is built. We can see that the graph is cyclic, and therefore,

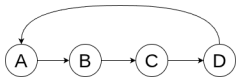

Fig. 7. Cyclic precedence graph

the order of the classes is inconsistent, and a linearization cannot be produced. There must exist an element such that, for every list in a list of lists, this element is **not in the tail** of the list, and it is **equal to the head of at least one** of those lists. in the tail of the list, and it is equal to the head of at least one of those lists.To detect the acyclicity of a graph, the following predicate was written:

```
64      (*@ predicate acyclic_precedence_graph (lins: C.t list list) =
65        forall c.
66          not (
67            exists path: C.t list.
68              List.length path > 1 /\
69              Sequence.get path 0 = c /\
70              Sequence.get path (List.length path - 1) = c /\
71              (forall i.
72                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 that give use to this predicate are not proven in this project and are assumed by postconditions and preconditions of the `merge` and `c3_linearization` functions.

4.4 merge function

The merge function is the core of the C3 linearization algorithm, and it is responsible for merging the lists of classes respecting the order of the classes in each list. The function is defined as follows:

```
    (*@ l = merge lins
          requires lins <> []
          requires forall i. 0 <= i < List.length lins ->
            distinct (Sequence.get lins i)
          requires acyclic_precedence_graph lins

          ensures distinct l
          ensures forall ia ib. 0 <= ia < ib < List.length l ->
            let ea = Sequence.get l ia in
            let eb = Sequence.get l ib in
            exists ja jb lin.
              Sequence.mem lin lins /\ Sequence.mem ea lin /\
              Sequence.mem eb lin /\ ea = Sequence.get lin ja /\
              eb = Sequence.get lin jb /\ ja < jb
          ensures forall e. not (Sequence.mem e l) ->
            forall lin. Sequence.mem lin lins ->
              not (Sequence.mem e lin)
          ensures forall lin e.
            (Sequence.mem lin lins) /\ not (Sequence.mem e lin) ->
            not (Sequence.mem e l)
          ensures forall e. Sequence.mem e l ->
            exists lin. (Sequence.mem lin lins) /\ (Sequence.mem e lin)
          variant sum_lengths lins
    *)
```

The function receives a list of lists, then for each head of the list is checked if it does not belong to any tail of the lists, if that condition is satisfied the head is removed from all the lists using the `remove` function, and the head is added to the result list.

Termination The termination of the function (line 385) is ensured by the postcondition of the `remove` function, which states that the sum of the lengths of the lists decreases with each call, so if the variant is the sum of the lengths of the lists, then the function will terminate.

Remove preconditions The preconditions of `remove`, states that for each call the element removed has to satisfy the condition previously described. To prove that, is required that the list of lists have a precedence graph acyclic (line 367) and this ensures, has discussed in the section 4.3, that the element removed is not in the tail of any list. This is stated by the following unproved lemma:

```
198        (*@ lemma acyclic_has_head_candidate:
199           forall lins: C.t list list, c: C.t.
200            acyclic_precedence_graph lins ->
201             (exists lin h t. Sequence.mem lin lins /\ Cons h t = lin /\
202             is_candidate_valid h lins)
203             *)
```

Main postconditions The two main postconditions of the `merge` function ensures that:

- The result is a list of classes that respects the order of the classes in each list of the input (lines 370 to 376).
- All the result classes present in the final list must exist in one of the lists of the input (lines 377 to 384).

Even assuming the acyclicity lemmas, this proof has yet to be proven.

Why3 IDE result In Fig. 8, we can see the proof progress of the `merge` function in the Why3 IDE.

4.5 c3_linearization function

The `c3_linearization` function differs from the original one to help the proof of termination, a new argument `universe` was added, which contains all the classes above in the hierarchy of the class. In that way, every time the parents of a class are called in the function `linearize` (an auxiliary function of `c3_linearization`), the class is removed from the universe, and the parents of the class are called with the universe as an argument. This ensures that the hierarchy remains acyclic and because the universe is always decreasing at each loop, it can be passed as argument and termination is ensured. Aside from the termination, the function has proved to be consistent with `merge` requirements. Finally, all the goals and their verification can be seen in Fig. 9.

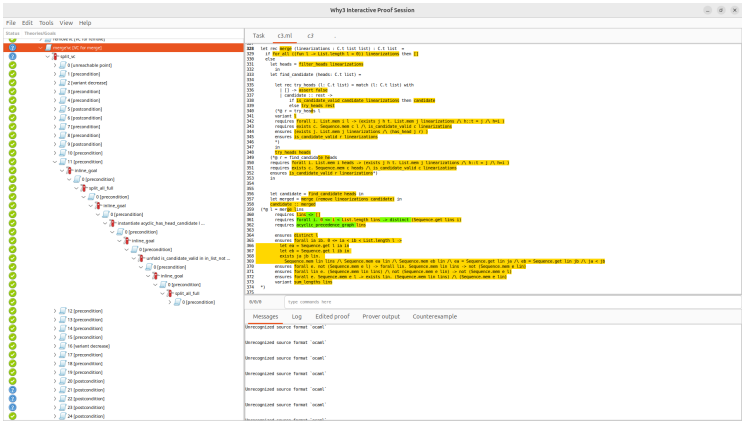

Fig. 8. `merge` proof in Why3 IDE

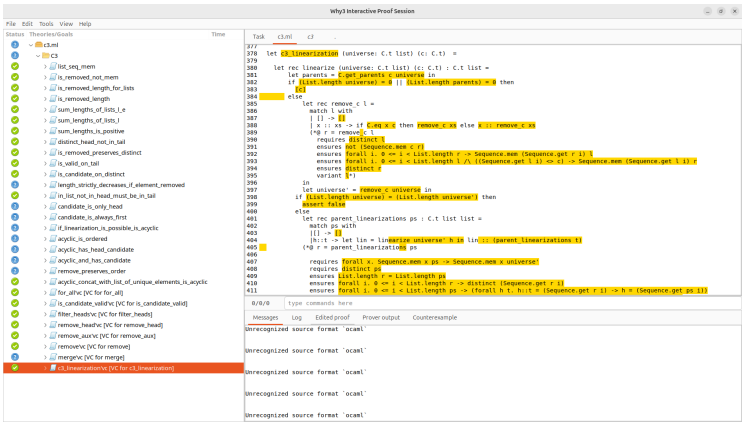

Fig. 9. All goals in Why3 IDE

5 Related Work

While the C3 linearization algorithm is widely used in programming languages, there has been limited formal verification of its implementation. Most existing studies focus on informal descriptions or implementations in specific languages, such as Python [3] and Perl [2]. To our knowledge, no verified implementation of the C3 linearization algorithm exists in OCaml or any other language.
There is a formal specification of the algorithm [15], however, it has a lot of errors and inconsistencies. Nevertheless, it was a good starting point for our work.
There is a verified implementation [16] of the OCaml library OCamlgraph [17]. This verification uses a similar approach to detect acyclicity in graphs and uses the same technology as our work, Why3 and Cameleer.
To get extract the accurately C3 properties, we used the concepts described by the work of Florent Hivert and others [6], which provides a detailed explanation of the C3 linearization algorithm and its properties.
Finally we used the work of Fawzi Albalooshi and others [18] to explore the effects of the multiple inheritance in programming languages, how it affects the MRO algorithms and how the diamond problem is resolved.

6 Conclusion and Future work

In this work, we addressed the need for a verified implementation of the C3 linearization algorithm, which is widely used in programming languages to resolve conflicts in multiple inheritance scenarios. We presented a mathematical formalization of the algorithm and initiated its verification using OCaml, Why3, and Cameleer. Our work proves termination and partially verifies the core components of the algorithm.

While the complete verification of the linearization process is still in progress, our contributions mark an important step toward a full proof of C3's correctness.

To complete the verification, we need to prove the acyclicity of the precedence graph. As discussed in Section 5, existing work on verifying acyclicity in Why3 and Cameleer provides a promising starting point for this task.

We also need to prove a lemma essential for the termination of the `merge` function, shown below:

```
151     (*@ lemma length_strictly_decreases_if_element_removed:
152           forall l1 l2: C.t list, e:C.t.
153             distinct l1 /\ distinct l2 /\ Sequence.mem e l1 /\
154             not (Sequence.mem e l2) /\
155             (forall x. Sequence.mem x l2 -> Sequence.mem x l1) /\
156             (forall x. Sequence.mem x l1 /\
157             x <> e -> Sequence.mem x l2) ->
158             List.length l2 < List.length l1*)
```

We experimented with `Sequence.get` (access by index) instead of `Sequence.mem` (membership check) to support this proof, but this was not sufficient to establish the lemma or to prove the termination of the `merge` function.

Finally, we aim to verify the `c3_linearization` function. This can be done by expressing as postconditions the properties defined in Section 3.2. To facilitate this, instead of using the `get_parents` function (Section 4.1) with a universe as input, the algorithm should be passed a class hierarchy directly as an argument. This could follow the type signature `(t * t list) list`, where the `t list` represents the parents of class `t`.

References

[1] 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. 1, 1996, pp. 69–82. ISBN: 978-0-89791-788-9. DOI: 10.1145/236337.236343. URL: https://dl.acm.org/doi/10.1145/236337.236343 (visited on 06/25/2025).

[2] *Mro - Method Resolution Order - Perldoc Browser*. URL: https://perldoc.perl.org/mro (visited on 06/25/2025).

[3] *The Python 2.3 Method Resolution Order*. Python documentation. URL: https://docs.python.org/3/howto/mro.html (visited on 06/25/2025).

[4] *Language Influences — Solidity 0.8.31 Documentation*. URL: https://docs.soliditylang.org/en/latest/language-influences.html (visited on 06/25/2025).

[5] *Miguelflor/C3_ocaml_verified*. URL: https://github.com/miguelflor/c3_ocaml_verified (visited on 07/04/2025).

[6] Florent Hivert and Nicolas M. Thiéry. "Controlling the C3 Super Class Linearization Algorithm for Large Hierarchies of Classes". In: *Order* 41.1 (Apr. 1, 2024), pp. 83–98. ISSN: 1572-9273. DOI: 10.1007/s11083-022-09607-5. URL: https://doi.org/10.1007/s11083-022-09607-5 (visited on 06/25/2025).

[7] 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. 30, 2021), pp. 020–031. ISSN: 25825003. DOI: 10.30574/gjeta.2021.8.3.0119. URL: https://gjeta.com/content/studies-object-oriented-programming-backbone-implementations (visited on 06/25/2025).

[8] 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.

[9] 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.

[10] 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.

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

[12] *Z3Prover/Z3*. Z3 Theorem Prover, July 3, 2025. URL: https://github.com/Z3Prover/z3 (visited on 07/03/2025).

[13] *About Cvc5*. cvc5. URL: https://cvc5.github.io/ (visited on 07/03/2025).

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

[15] João Carlos Raposo dos Reis. "Towards a Solider Solidity". MA thesis. Dec. 2023. URL: https://run.unl.pt/handle/10362/167656 (visited on 07/04/2025).

[16] Daniel Castanho and Mário Pereira. *Auto-Active Verification of Graph Algorithms, Written in OCaml*. July 20, 2022. DOI: 10.48550/arXiv.2207.09854. arXiv: 2207.09854 [cs]. URL: http://arxiv.org/abs/2207.09854 (visited on 07/04/2025). Pre-published.

[17] Jean-Christophe Filliatre. *Backtracking/Ocamlgraph*. June 23, 2025. URL: https://github.com/backtracking/ocamlgraph (visited on 07/04/2025).

[18] Fawzi Albalooshi and Amjad Mahmood. "A Comparative Study on the Effect of Multiple Inheritance Mechanism in Java, C++, and Python on Complexity and Reusability of Code". In: *International Journal of Advanced Computer Science and Applications* 8.6 (2017). ISSN: 21565570, 2158107X. DOI: 10.14569/IJACSA.2017.080614. URL: http://thesai.org/Publications/ViewPaper?Volume=8&Issue=6&Code=ijacsa&SerialNo=14 (visited on 06/25/2025).