# A Compositional Deadlock Detector for Android Java

Darius Muresan

Henrique Luz
Rui Xavier

Aarhus University
Department of Computer Science

November 11, 2025

AARHUS UNIVERSITY

# Why Study Deadlocks?

- Deadlocks are a fundamental challenge in concurrent programming.
- They occur when threads cyclically wait on each other's locks, preventing further progress.
- Detecting and preventing them is critical for reliability, especially in large-scale systems.
- In industrial settings (e.g., Android applications at Facebook), codebases can exceed tens of millions of LoC, making traditional whole-program analyses infeasible.

# The Industrial Challenge

- Modern software development involves:
  - Massive, continuously evolving codebases.
  - Frequent commits and rapid code reviews.
  - Strong requirements for developer feedback in under 15 minutes.
- Conventional static analyses:
  - Require analyzing entire programs.
  - Are too slow or memory-intensive for this context.
  - Often produce many false positives.

## Goal

Enable fast, scalable, and accurate deadlock detection that integrates seamlessly into continuous integration pipelines.

# The Research Gap

- Most existing tools:
  - Lack compositionality by reanalyzing the whole program each time.
  - Sacrifice scalability for soundness.
- Desired: a compositional, mathematically grounded approach that can analyze only changed code and its dependents.

## Core Challenge

How can we design a deadlock analysis that is both **theoretically sound** and **practically scalable** to millions of lines of concurrent Android Java code?

# Our Research Question

## Research Question

Can we develop a **compositional deadlock detector** for **Android Java** that is sound, complete, and efficient enough to run on commercial-scale codebases under active development?

- Key aspects explored:
  - Abstract modeling of real Java concurrency (balanced, re-entrant locks).
  - Deadlock characterization via **critical pairs**.
  - Compositional analysis integrated into the **INFER** framework.
- Goal: bridge the gap between **theory** and **industrial-scale deployment**.

# Concurrent Programs

# How to look at Concurrent Programs?

*Gospel + OCaml*

```
type α node = {
  data : α;
  mutable prev : α node;
  mutable next : α node;
}
```

*CFML*

```
Definition Node A (v: A) (n p c: loc) : hprop :=
c ↦ `{ data' := v; next' := n; prev' := p }.
```

# Inner Node

```
type α innerNode =
  | Nil
  | Cons of α node
```

*Gospel + OCaml*

```
Definition InnerNode A (L: list A) (p: innerNode_ A) : hprop :=
  match L with
  | [] ⟹ [p = Nil]
  | _ ⟹ ∃ (c q: loc), [p = Cons c] ⋆ c ⇝ NodeSeg L c q q
  end.
```

*CFML*

# Doubly Linked List

*Gospel + OCaml*

```
type α dblist = {
  mutable head : α innerNode;
  mutable tail : α innerNode;
  mutable length : int;
}
```

*CFML*

```
Definition Dblist A (L: list A) (l: loc) : hprop :=
  ∃ (p q: innerNode_ A),
      (l ↦ `{ head' := p; tail' := q; length' := length L }) ⋆
      (If L = [] then [p = Nil] ⋆ [q = Nil]
       else
         ∃ x L' h t, [L = x :: L'] ⋆ [p = Cons h] ⋆ [q = Cons t]
                            ⋆ (h ⇝ NodeSeg L h t t)).
```

# Program Executions and their traces

*GOSPEL + OCaml*

```
val create : unit → α dblist
val is_empty : α dblist → bool
val clear : α dblist → unit
val remove_head : α dblist → α node
val remove_tail : α dblist → α node
val reverse : α dblist → unit
val append : α dblist → α dblist → α dblist
val josephus : α dblist → int → unit
val fold_right : (α → β → β) → α dblist → β → β
val fold_left : (α → β → α) → α → β dblist → α
val iter_right : (α → β) → α dblist → β
val iter_left : (α → β) → α dblist → unit
...
```

*GOSPEL + OCaml*

```
val create : unit → α dblist
val is_empty : α dblist → bool
val clear : α dblist → unit
val remove_head : α dblist → α node
val remove_tail : α dblist → α node
val reverse : α dblist → unit
val append : α dblist → α dblist → α dblist
val josephus : α dblist → int → unit
val fold_right : (α → β → β) → α dblist → β → β
val fold_left : (α → β → α) → α → β dblist → α
val iter_right : (α → β) → α dblist → β
val iter_left : ('a -> 'b) -> 'a dblist -> unit
...
```

# Predicates

Filliâtre and Pereira [6] introduced two key logical predicates, *permitted* and *complete*, which allow us to reason both about the correctness of an iterator's behavior and its termination.

## Permitted – Visited Elements

$$permitted(v, s) \triangleq ||v|| \leq ||s|| \land \forall i : \ 0 \leq i \leq ||v|| \implies v[i] = s[i]$$

## Complete – Iteration Termination

$$complete(v, s) \triangleq ||v|| = ||s||$$

# iter_left Function

*Gospel + OCaml*

```
val iter_left : (α → β) → α dblist → unit
(*@ r = iter_left f collection
  iterspec
  ~permitted: (fun v → length v ≤ length collection ∧
               ∀ i. 0 ≤ i < length v → v[i] = collection[i])
  ~complete:  (fun v → length v = length collection) *)
```

This specification is a recent addition to Gospel by Ion Chirica and Mário Pereira [3].

# Soundness and Completeness

# OCaml Code

*OCaml*

```
let rec iter_left_aux f node tail =
  f node.data;
  if not (tail == node) then
    iter_left_aux f node.next tail

let iter_left f db =
  match db.head with
  | Nil → ()
  | Cons n →
    match db.tail with
    | Nil → assert false
    | Cons tail → iter_left_aux f n tail
```

# Consumer Function Specification

```
∀ x L1 L2, L = L1 ++ x :: L2 →
      SPEC (f x)
        PRE (I L1)
        POSTUNIT (I (L1 & x))
```

*CFML*

# iter_left Specification

```
Lemma Triple_iter_left : ∀ (I: list A → hprop) L (f: val) t,
    (∀ x L1 L2, L = L1 ++ x :: L2 →
      SPEC (f x)
        PRE (I L1)
        POSTUNIT (I (L1 & x))) →
    SPEC (iter_left f t)
      PRE (t ⇝ Dblist L ⋆ I [])
      POSTUNIT (t ⇝ Dblist L ⋆ I L).
```

*CFML*

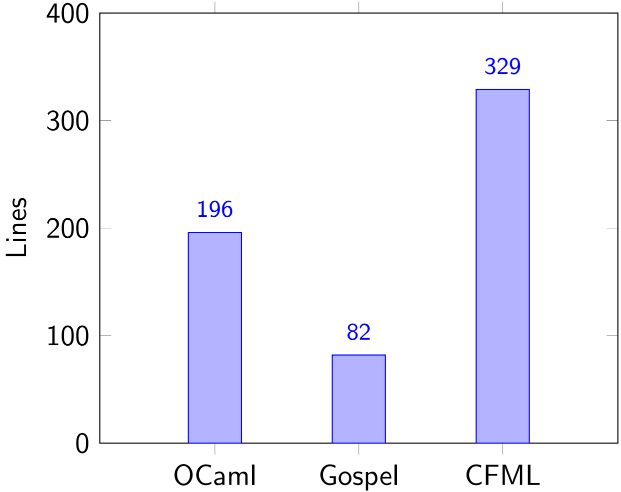
# iter_left_aux Specification

To verify the higher-level `iter_left` function, we require a more fine-grained specification for its auxiliary, lower-level implementation.

*CFML*

```
Lemma Triple_iter_left_aux :
  ∀ (f: val) (I : list A → hprop) (L L1 L2: list A) n e h p,
    (∀ x L1 L2, L = L1 ++ x :: L2 →
                SPEC (f x)
                  PRE (I L1)
                  POSTUNIT (I (L1 & x))) →
    L = L1 ++ L2 →
    L2 ≠ nil →
    SPEC (iter_left_aux f n e)
      PRE  (n ⇝ NodeSeg L2 h e p ⋆ I L1)
      POSTUNIT (n ⇝ NodeSeg L2 h e p ⋆ I L).
```

# Upper Complexity Bounds



400
300
200
100
0
Lines
196
82
329
OCaml
Gospel
CFML

# Implementation

# Conclusion and Related Work

# Conclusions

- Designed and implemented a circular doubly linked list in OCaml.
- Verified key operations (such as `create`, `clear`, `iter_left`, `fold_left`) using CFML.
- Demonstrated feasibility of verifying mutable data structures in a functional language.
- Identified limitations within the current toolchain.

# Reflections & Future Work

## Reflections

- Formal verification remains time-intensive (some proofs may take months).
- Highlights both promise and current challenges in formally verifying real-world functional code.

## Future Work

- Verify the remaining operations and auxiliary lemmas.
- Perform the refinement by comparing manual specs with Peter-generated ones.

# Bibliography I


[1] Arthur Charguéraud. "Characteristic formulae for the verification of imperative programs". In: *SIGPLAN Not.* 46.9 (Sept. 2011), pp. 418–430. ISSN: 0362-1340. DOI: 10.1145/2034574.2034828. URL: https://doi.org/10.1145/2034574.2034828.

[2] Arthur Charguéraud et al. "GOSPEL - Providing OCaml with a Formal Specification Language". In: *Formal Methods - The Next 30 Years - Third World Congress, FM 2019, Porto, Portugal, October 7-11, 2019, Proceedings*. Ed. by Maurice H. ter Beek, Annabelle McIver, and José N. Oliveira. Vol. 11800. Lecture Notes in Computer Science. Springer, 2019, pp. 484–501. DOI: 10.1007/978-3-030-30942-8_29. URL: https://doi.org/10.1007/978-3-030-30942-8%5C_29.

# Bibliography II


[3] Ion Chirica and Mário Pereira. "Unfolding Iterators: Specification and Verification of Higher-Order Iterators, in OCaml". In: *arXiv preprint arXiv:2506.20310* (2024). DOI: 10.48550/arXiv.2506.20310. arXiv: 2506.20310 [cs.PL]. URL: https://arxiv.org/abs/2506.20310.

[4] Michael R. Clarkson et al. *OCaml Programming: Correct + Efficient + Beautiful*. https://cs3110.github.io/textbook/cover.html. 2025.

[5] Jean-Christophe Filliâtre and Clément Pascutto. "Ortac: Runtime Assertion Checking for OCaml (Tool Paper)". In: *Runtime Verification: 21st International Conference, RV 2021, Virtual Event, October 11–14, 2021, Proceedings*. Berlin, Heidelberg: Springer-Verlag, 2021, pp. 244–253. ISBN: 978-3-030-88493-2. DOI: 10.1007/978-3-030-88494-9_13. URL: https://doi.org/10.1007/978-3-030-88494-9_13.

# Bibliography III


[6] Jean-Christophe Filliâtre and Mário Pereira. "A Modular Way to Reason About Iteration". In: *NASA Formal Methods - 8th International Symposium, NFM 2016, Minneapolis, MN, USA, June 7-9, 2016, Proceedings*. Ed. by Sanjai Rayadurgam and Oksana Tkachuk. Vol. 9690. Lecture Notes in Computer Science. Springer, 2016, pp. 322–336. DOI: 10.1007/978-3-319-40648-0_24. URL: https://doi.org/10.1007/978-3-319-40648-0%5C_24.

[7] Marc Hermes and Robbert Krebbers. "Modular Verification of Intrusive List and Tree Data Structures in Separation Logic". In: *15th International Conference on Interactive Theorem Proving (ITP 2024)*. Ed. by Yves Bertot, Temur Kutsia, and Michael Norrish. Vol. 309. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024, 19:1–19:18. ISBN: 978-3-95977-337-9. DOI: 10.4230/LIPIcs.ITP.2024.19. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ITP.2024.19.

# Bibliography IV


[8] Mário Pereira and António Ravara. "Cameleer: A Deductive Verification Tool for OCaml". In: July 2021, pp. 677–689. ISBN: 978-3-030-81687-2. DOI: 10.1007/978-3-030-81688-9_31.

[9] Mário Pereira Rui Xavier and Tiago Soares. *Formal Verification of OCaml Programs with Dynamic Memory*. Companion Artifact. 2025. URL: https://github.com/RuiXavier/adc-artifact-companion (visited on 06/23/2025).

[10] Tiago Lopes Soares, Ion Chirica, and M'ario Pereira. "Static and Dynamic Verification of OCaml Programs: The Gospel Ecosystem". In: *Leveraging Applications of Formal Methods, Verification and Validation. Specification and Verification*. Ed. by Tiziana Margaria and Bernhard Steffen. Cham: Springer Nature Switzerland, 2025, pp. 247–265. ISBN: 978-3-031-75380-0.